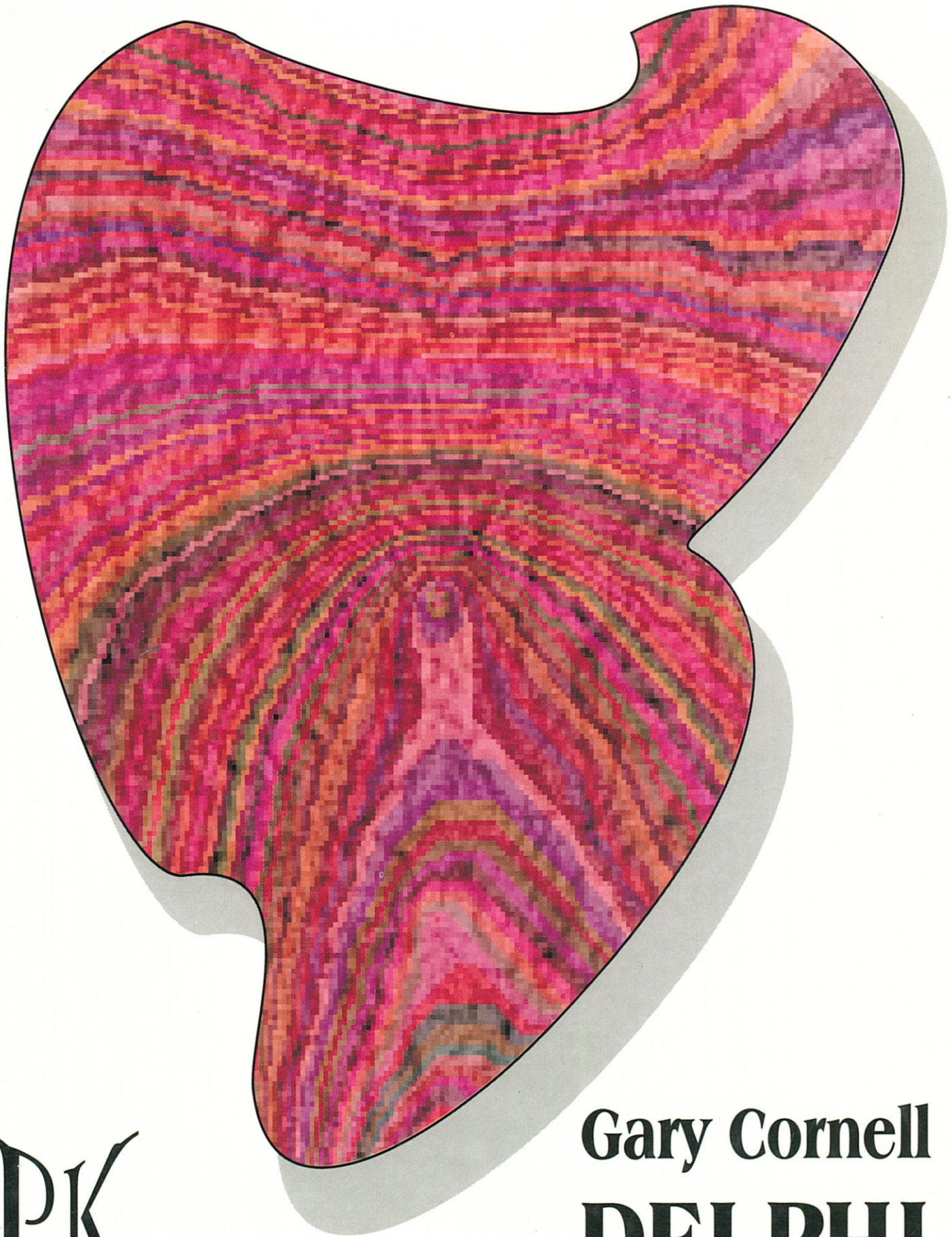


PK

PROGRAMOZÓK
KÖNYVTÁRA

Gary Cornell

DELPHI

Tippek és trükkök

PROGRAMOZÓK
KÖNYVTÁRA

Gary Cornell

Delphi

Tippek és trükkök

Az 1. és 2. verzióhoz a 16, illetve 32 bites fejlesztésekhez

PANEM–McGRAW-HILL

A mű eredeti címe: Delphi Nuts & Bolts: For Experienced Programmers, Second Edition.
Osborne McGraw-Hill, 2600 Tenth Street, Berkeley, California 94710, U. S. A.
Copyright © 1996 by McGraw-Hill, Inc.

Copyright © Hungarian edition Panem Kft. – McGraw-Hill Inc., 1997

McGraw-Hill Book Company Europe
Schoppenhangers Road
Maidenhead, Berkshire, SL6 2QL
England

Panem Kft.
1385 Budapest, Pf. 809
Hungary

Ez a könyv a Művelődési és Közoktatási Minisztérium támogatásával
a Felsőoktatási Pályázatok Irodája által lebonyolított
felsőoktatási tankönyvtámogatási program keretében jelent meg.

ISBN 963 545 124 5

A kiadásért felel a Panem Kft. ügyvezetője, Budapest, 1997
Fordította: Morvay Gábor
Lektorálta: Berzsenyi Miklós
Műszaki szerkesztő: Érdi Júlia
Borítóterv: Érdi Júlia

A Panem könyvek megrendelhetők a 06-30/488-488 hívószámú mobiltelefonon,
illetve a 1385 Budapest, Pf. 809 levélcímen.
Email: 100324.513@compuserve.com
http://www.mcgraw-hill.co.uk/Panem/

Minden jog fenntartva. Jelen könyvet, ill. annak részeit tilos reprodukálni, adatrögzítő rendszerben tárolni, bármilyen formában vagy eszközzel – elektronikus úton vagy más módon – közölni a kiadók engedélye nélkül.

Tartalom

BEVEZETÉS 13

1. ALAPOK 19

Miért éppen Windows és miért pont Delphi? 19
Hogyan fejleszthetünk Delphi alkalmazást? 21
Hogyan dolgozzunk a delphivel? 25
Egy „Helló Világ" típusú program 27
Egy nagyobb projekt készítése 34

2. A DELPHI 32 PROGRAMOZÓI KÖRNYEZET 39

A fő képernyőfelület áttekintése 40
A fejléc 41
A főmenü 42
Az eszközpaletta 43
A komponenspaletta 45
A kiindulási formablak 46
Az objektumfelügyelő ablaka 46
A kódszerkesztő ablak 47
A Project Manager ablak 48
Az Alignment Palette (elrendezéskezelő paletta) 49
A súgórendszer 49
A file menüről részletesebben 50
Szövegszerkesztés 53
Az Edit Menü 54
A Search menü 56
A View menü 57

A Project menü 59
A Run menü 60
A Component menü 60
A Tools menü 61
Programok betöltése és futtatása 62

3. A DELPHI 16 PROGRAMOZÓI KÖRNYEZET 64

A fő képernyőfelület áttekintése 65
A fejléc 66
A főmenü 67
Az eszközpaletta 68
A komponenspaletta 70
A kiindulási formablak 70
Az objektumfelügyelő ablaka 70
A kódszerkesztő ablak 71
A Project Manager ablak 72
Az Alignment Palette (elrendezéskezelő paletta) 73
A súgórendszer 74
A file menüről részletesebben 75
Szövegszerkesztés 78
Az Edit Menü 78
A Search menü 80
A View menü 81
Az Options menü 83
A Tools menü 85
Programok betöltése és futtatása 86

4. FORMOK TERVEZÉSE: KOMPONENSEK, MENÜK, ESEMÉNYEK 88

Az objektumfelügyelő használata 89
Az egyszerű szerkesztő 90
Szerkesztés legördülő listával 91
A dialógusablak-szerkesztő 91
Beágyazott tulajdonságszerkesztő 92
A Color tulajdonság beállítása 93

Formok 95
Formok tulajdonságai 95
Formesemények 101
Komponensek 103
A leggyakoribb komponensek áttekintése 104
Komponensek illesztése a formra 108
A formon lévő komponensek manipulálása 111
Komponensek elrendezése 114
Néhány gyakori komponens áttekintése 115
Fókusszal kapcsolatos tulajdonságok és események 116
A TLabel (címke) komponens 117
A TEdit komponens 118
A TMemo komponens 119
A TButton és a TBitButton komponens 120
A TListBox és a TComboBox komponens 122
A TCheckBox és a TRadioButton komponens 124
A TTimer komponens 124
A TStringGrid és a TDrawGrid komponens 127
Menük tervezése 128
A menütervező használata 131
Formok ASCII reprezentációja 136

5. A DELPHI PROGRAMOZÁS ALAPJAI 139

A Delphi parancsai 141
Megjegyzések 142
Programok befejezése 143
Tulajdonságok beállítása 144
Logikai tulajdonságok 146
A with kulcsszó 147
Változótípusok 148
A Delphi 32 új típusai 152
Változók 156
Értékadás változóknak 158
Konstansok 159
A delphi eseménykezelők anatómiája 161
Változók és konstansok hatásköre 162

Eljárások közös változói ... 163
Statikus változók használata ... 165
Karakterláncok (sztringek) ... 167
Számok ... 168
Műveletek számokkal ... 169
Néhány szó még a Delphi számhasználatáról ... 173
Típuskonverziók ... 173
Iterált műveletvégrehajtás-ciklusok ... 174
Elöltesztelő léptető ciklus – for ciklus ... 174
Nem determinisztikus ciklusok ... 178
Feltételes utasításvégrehajtás ... 180
If-then utasítások használata ciklusokkal ... 182
A case utasítás ... 183
Objektumok kezelése futási időben ... 185
Metódusok ... 185
A TApplication objektum ... 187
A TScreen objektum ... 189
A TCanvas objektum ... 190
A TPrinter objektum ... 191
Üzenetablakok és inputablakok ... 194
Az InputBox függvény ... 198
Több formot tartalmazó alkalmazások ... 200
További formok illesztése Delphi 16 projektekhez ... 200
További formok illesztése Delphi 32 projektekhez ... 202
Több formot tartalmazó projektek kezelése ... 203
Formok mentése ... 203
Form minták használata ... 204
Form (és a hozzátartozó unit) törlése a projektből ... 204
A projekt főformjának kijelölése ... 205
Formok viselkedése futási idő alatt ... 206
Programozás formokkal ... 209

6. MAGASABB SZÍNTŰ PROGRAMOZÓI TECHNIKÁK ... 211

Újra a típusokról ... 211
Sorszámozott típusok ... 212
Halmazok ... 215

Tömbök ... 217
Egydimenziós tömbök ... 217
Sztringlisták ... 220
Többdimenziós tömbök ... 223
Rekordok ... 224
Mutatók ... 227
Mutatók és a memóriakezelés ... 231
Nulla-végű sztringek és a PChar ... 233
Függvények és eljárások ... 235
A beépített Runtime Library (RTL) ... 238
Felhasználó által definiált függvények és eljárások ... 242
Külső DLL-ek használata ... 248
Rekurzió ... 249
Projektek és unitok kezelése ... 252
Bővebben a unitok részeiről ... 253
Lényeges tudnivalók a fordítóról ... 256
A GOTO utasítás ... 262

7. OBJEKTUMOK ... 265

Bevezetés ... 266
Az OOP terminológiája ... 268
Objektumok létrehozása a delphiben ... 272
A visual component library objektumai ... 280
A formdeklaráció private és public részei ... 281
Delphi objektumok létrehozása futási idő alatt. ... 286
Új vizuális objektumok létrehozása futási időben ... 287
A Delphi objektumböngészője ... 288
A fordítási direktívák és hatásuk az objektumböngészőre ... 291
Objektumok mélyrehatóbb vizsgálata ... 292

8. HIBA- ÉS KIVÉTELKEZELÉS ... 293

Védett blokkok ... 294
A kivételhierarchia ... 296
Kivételosztályok kezelése ... 301
Kivételelek ismételt előidézése ... 301
Védett erőforrás-foglalások ... 302

Felhasználó által definiált kivételek 304
Kivételobjektumok deklarálása 304
Csendes kivételek 305

9. HIBAJAVÍTÁSI ESZKÖZÖK ÉS TECHNIKÁK 306

Hibakereső eszközök és feladatkörük 307
Hogyan készítsünk olyan programokat, amelyekben könnyű a hibákat megtalálni? 307
Logikai hibák 309
Nyomkövetési információk 310
Az evaluate/modify dialógusablak 311
Hogyan használjuk az Evaluate/Modify dialógusablakot nyomkövetésre? 312
A Trace into funkció 313
A program futásának felfüggesztése 314
A Watch funkció 315
Néhány végső megjegyzés a hibakeresésről 317
Eseményvezérelt hibák és problémák 318
Programozási stílus 318

10. FÁJLKEZELÉS 320

Könyvtár- és állománykezeléssel kapcsolatos parancsok 321
Fájlrendszer-komponensek 327
A FileListBox komponens 328
A DirectoryListBox komponens 330
A DriveComboBox komponens 330
A FilterComboBox komponens 330
A fájl komponensek összekötése 331
Szöveges állományok 331
Adatok beolvasása szövegesállományból 335
I/O ellenőrzés fájlkezelés során 336
Számok olvasása szöveges állományokból 337
Típusos állományok 339
Rekordokat tartalmazó típusos állomány 340
Típus nélküli állományok 341
Parancssor-paraméterek használata 343

11. WINDOWS ALKALMAZÁSOK KÖZTI ADATCSERE 344

A clipboard 346
Clipboard formátumok és grafikus átvitelek 348
Dinamikus adatkapcsolatok (DDE) 349
DDE kapcsolat létesítése fejlesztési időben 350
DDE kapcsolatok futási időben 352
DDE komponensek tulajdonságai, metódusai és eseményei 353
Objektumok csatolása és beágyazása (OLE) 358
Az OLE használata 361
Az OLE 2 használata fejlesztési időben 362
A Paste Special dialógusablak 364
A TOleContainer komponens tulajdonságai 364
OLE kapcsolatok létesítése programon keresztül 365
OLE Automation (OLE Automatizálás) 367

12. GRAFIKA 371

A TShape komponens 372
A grafika alapjai 373
A TImage komponensről részletesebben 375
A Canvas objektum alapjellemzői 375
Színek 375
Pixelgrafika 376
Alakzatok rajzolása 377
Rajzolás tollakkal 378
Ecsetek használata 379

13. PROFI FELHASZNÁLÓI FELÜLET KÉSZÍTÉSE 381

Eszközpaletta és állapotjelző sor beillesztése projektjeinkbe 382
Eszközpaletta és állapotjelző sor létrehozása 382
Általános dialógusablakok 385
Általános dialógusablakok kezelése 386
MDI formok 390
A Window menü – a gyermekablakok elrendezése 391
Ablakok megjelenítése – Z-sorrend 392
Egérműveletek nyomonkövetése 393
Az egérrel kapcsolatos eseménykezelők 394

Fogd és vidd műveletek ... 398
Súgórendszerek ... 403
Windows 95 vezérlőelemek ... 407
A TRichEdit komponens ... 408
A helyesírás-elemző OCX ... 409

14. ADATBÁZIS-KEZELÉSI LEHETŐSÉGEK ... 410

Néhány szó a modern adatbázisokról ... 411
A Delphi adatbázis architektúrája ... 413
A Data Access komponensek ... 413
A Data Controls oldal komponensei ... 416
Database form expert ... 418
Végső megjegyzés ... 419

Bevezetés

Jó ideje nem jelent meg ilyen izgalmas nyelvi termék a piacon. A sajtó ritkán látott lelkesedéssel fogadta a Delphit. Miért ez a nagy csinnadratta? Mi is pontosan ez a Delphi, és mire jó? Nos, röviden azt lehet mondani, hogy a Delphi nem más, mint egy egyszerűen kezelhető, mégis rendkívül hatékony eszköz Windows alkalmazások fejlesztéséhez. Erősségei közé tartozik az a modern fordító technika, amellyel hihetetlenül gyors, önmagában (értelmező rendszer nélkül) futtatható programokat hozhatunk létre. Lehetőséget kínál valódi objektumorientált programozásra és kliens-szerver adatbázis fejlesztésre is.

Ha ez kevésnek tűnne a Delphi körüli felhajtás megindoklásához, akkor érdemes még megemlíteni, hogy a legbonyolultabb programok fejlesztésénél is csak igen ritkán van szükség C vagy C++ jártasságra a Delphi használata mellett. Mostantól nem kell, hogy profi C programozók legyünk és íróasztalunkon kb. 10 kilónyi dokumentáció heverjen ahhoz, hogy villámgyors Windows alkalmazásokat fejleszthessünk.

A könyvről

Gyakorlott programozók ezt a könyvet segédanyagnak tekinthetik a Delphi mindkét verziójához. A könyvet annak a reményében írtam, hogy egyformán hasznos lesz Windows 3.X-ben és Windows 95-ben fejlesztő programozók számára. Az olvasóról csak annyit feltételezek, hogy rendelkezik valamennyi programozói (mindegy milyen nyelven szerzett) tapasztalattal, és tisztában van olyan fogalmakkal, mint ciklus és döntési struktúra. Összefoglalva a célom az, hogy megismertessem az olvasót a

Delphi képességeivel, és azzal, hogy miért szükséges, hogy a Delphi programozói eszköztárunk részévé váljon.

Szándékom a könyv megírásával az volt, hogy a Delphi mindkét verziójának a lényegét kiszűrjem, és itt rövid és érthető formában tálaljam. Mindvégig próbáltam hangsúlyozni a teljesen új gondolkodásmódot, amely nélkülözhetetlen a Delphivel történő programozás elsajátításához, így más programnyelvekben jártas fejlesztők is profitálhatnak a könyvből. Ezt a tárgyalási formát tartottam jobbnak, mert a Delphi más programnyelvek kliséibe történő beleerőltetése kudarcra van ítélve – nem lehet igazán kiaknázni a lehetőségeit, ha nem tudunk kitörni a beidegződött megközelítési módokból.

Feltételezésem szerint a Delphi „motorjának" megismerése után könnyen elboldogulhat a beépített súgó és a Delphihez járó kézikönyv segítségével. Nem célom sok példaprogrammal terhelni az olvasót, aki – feltételezem – elegendő programot írt már ahhoz, hogy követni tudja a könyvet. Megítélésem szerint, ha már elsajátította ezen új fejlesztői környezet által megkövetelt gondolkodásmódot, a Delphihez mellékelt példaprogramok között találhat olyat, amelyekből még tanulni lehet, ha a közöltek kevésnek bizonyulnak.

Hogyan olvassuk a könyvet?

Igényeinktől és hátterünktől függően a könyvet sokféleképpen olvashatjuk. Azok, akik jártasak a Visual Basic használatában, lehet hogy beérik az első néhány fejezet gyors átolvasásával, és később kapcsolódnak igazán a könyvhöz, ahol a nyelv alapjaival ismerkedhetnek meg. A Pascal vagy C nyelv ismerői dönthetnek úgy, hogy inkább az első fejezetekre szánnak több időt és a nyelvi alapokat ismertető részeket gyorsan olvassák át. (Mindenképpen célszerű átnézni minden fejezetet, mert azok fontos tudnivalókat tartalmaznak a Delphiről.)

A 14 fejezet rövid tartalma:

- Az 1. fejezet bevezetés a Delphibe, és itt található két egyszerű program is.
- A 2. fejezet bemutatja a Delphi 32 környezetét.
- A 3. fejezet a Delphi 16 környezetét ismerteti.

- A 4. fejezet megismerteti az olvasót a testreszabható ablakokkal (a formokkal), amelyek minden Microsoft Windows alatti alkalmazás lelke (így a Delphinek is). Megtanulhatjuk, hogyan adhatjuk a kívánt formhoz a Delphi alkotóelemeit, úgymint nyomógombokat, szövegmezőket, cimkéket, időzítőket és pozicionálórácsokat és hogyan lehet ezeket igény szerint alakítani.
- Az 5. és a 6. fejezetben áttekintjük a Delphiben használt Object Pascalt.
- A 7. fejezet rövid bevezetésként szolgál az objektumorientált programozásba.
- A 8. fejezetben megismerkedhetünk azokkal a Delphi nyújtotta bravúros technikákkal, ahogy a futás közben fellépő programhibákat (kivételeket=**exceptions**) kezelni lehet.
- A 9. fejezet felöleli a hibakeresési technikákat.
- A 10. fejezet a fájlkezelő technikákkal ismerteti meg az olvasót.
- A 11. fejezet bevezetés a DDE (dynamic data exchange=dinamikus adatkapcsolatok) és az OLE (object linking and embedding= objektumok csatolása és beágyazása) világába.
- A 12. fejezet a grafikai lehetőségekbe vezeti be az olvasót. Ez fontos, hiszen a Microsft Windows egy grafikus környezet. A Delphi erőforrásai ezen a téren meglehetősen látványosak.
- A 13. fejezet több magasszintű felhasználói felület (advanced user interface), többek között egérkezeléssel kapcsolatos technikák bemutatását öleli fel.
- A 14. fejezet rövid bevezetés a Delphi adatbázis-kezelő lehetőségeibe.

Megjegyzés: *Ennek a könyvnek nem célja, hogy bemutassa, hogyan lehet kliens-szerver alkalmazásokat fejleszteni a Delphi segítségével.*

A könyvben használt konvenciók

Először is a Delphi 32 bites verzióját az elterjedtebb „utcai" nevén Delphi 32-nek fogom hívni a hivatalos (ritkábban használt) Delphi 2.0 helyett. Hasonlóképpen a hivatalos Delphi 1.0 -nak a Delphi 16 felel majd meg. A Windows 95 előtti Windows változatokra a Windows 3.X fog utalni.

A billentyűk neveinek kisméretű nagybetűs írásmódot választottunk. Például a CRT és ALT billentyűk a későbbiekben úgy jelennek meg, ahogy az itt is látható. A nyilak és más vezérlő karakterekre hivatkozás teljes nevükön, szintén kisméretű nagybetűs formátumban történik. Például, ha a jobbra nyilat kell használnunk, a következőt fogjuk látni: „Nyomja le a JOBBRA NYILAT!".

Ha bizonyos menüpontok aktivizálásához billentyűkombinációk használata szükséges, akkor a lenyomandó billentyűk egymástól kötőjellel elválasztva jelennek meg a szövegben, és az egész kombináció kisméretű nagybetűs formátumban szerepel majd. Így például a „Nyomja meg a CTRL-A-B-t" azt jelenti, hogy lenyomva tartjuk a „CTRL" billentyűt, miközben lenyomjuk először az „A" majd a „B" gombot. Másfelől viszont az „ALT-F, P" azt fogja jelenteni, hogy először az „ALT" és „F" gombokat nyomjuk le egyszerre, majd külön a „P"-t – az „ALT" lenyomása nélkül.

DOS parancsok, fájlnevek, és fájlok kiterjesztései nagybetűs írásmódban jelennek meg: COMMAND.COM, .TXT stb. Megpróbálom az eredeti Delphi dokumentáció konvencióit követni. A menüpontokban az egymás utáni választásokat függőleges vonalakkal választom el egymástól: pl. „Válasszuk a Run | Run pontot". A kulcsszavak vastag kisbetűs formában szerepelnek (pl. „**try/finally**"). A könyvtári függvények és eljárások a Delphiben láthatóknak megfelelően vannak feltüntetve, azaz általában minden szó nagy kezdőbetűvel, úgymint Font, TextOut stb.

A Delphi parancsok szintaxisa többnyire hagyományos szöveg formában, egysoros listaként van szedve. A szintaxis programozó által megadható részeit dőltbetűs formában írt kifejezések jelzik. A fájlok átnevezésére használt Rename parancs szintaxisa az előbbiek alapján így néz ki:

Rename (*RégiÁllományNév,ÚjÁllományNév*)

A mintaprogramok kisméretű, vékony karakteres formátumban vannak szedve a következőképpen:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  ShowMessage('Hello world!');
end;
```

(Habár a Delphi a kulcsszavakat kövérbetűs formában jeleníti meg, ezt én csak az első fejezetben vagy programvázak közlésekor teszem meg.)

A számtalan tippet és megjegyzést különböző ikonok kísérnek, amelyeknek figyelemfelkeltő szerepe van. Például egy megjegyzés a következőképpen néz ki:

Megjegyzés: *A képek nagyrésze Windows 95 alatt futó Delphi 32-ből származik. A Delphi 16 képernyőfelületei nagyon hasonlóak.*

Egy általános tipp így néz ki:

Tipp: *Ne telepítsük mindkét Delphi változatot ugyanabba a könyvtárba!*

Vannak olyan tippek, amelyek főként Visual Basic programozóknak szólnak. Ezek így néznek ki:

Visual Basic tipp: *Delphi 16-ot egy a DOC alkönyvtárban található DELPHI2VB.TXT nevű fájllal együtt forgalmazzák, amelyet érdemes tanulmányoznunk, ahogy egy kis jártasságot szereztünk a Delphiben.*

Végül, azoknak, akik a Borland's Pascal korábbi verziójáról váltottak a Delphire, szól a következőképpen megjelenő tipp:

Pascal tipp: *Ne próbálkozzunk a Borland's Object Vision-t használó programok és a Delphi's Visual Component Library kombinálásával!*

1. FEJEZET

Alapok

Ennek a fejezetnek a célja, hogy megismertesse az olvasót a Delphi 16 és 32 bites verziójában történő programozás fogalmaival. Itt tapasztalja először, hogy milyen könnyen lehet egyszerű Delphi programokat írni. A fejezet végén közölt két program bevezetésként szolgál az eseményvezérelt, objektumorientált modellbe, amely mindkét Delphi verzió hatékonyságának alapja. A fejezetben nem kerül bemutatásra semelyik Delphi változat telepítésének módja, elindítása, az ablakok kezelése Windows környezetben stb. (ezeket mind ismertnek tekintjük). Nem feltételezünk azonban semmiféle előképzettséget a Windows és a Delphi programozásában (habár egy bizonyos szintű programozási tudás – mindegy, milyen környezetben szerzett – szükséges).

■ *Miért éppen Windows és miért pont Delphi?*

A grafikus felhasználói felületek (Graphical User Interfaces), vagy GUI-k, forradalmasították a mikroszámítógép-ipart. Ezek jól demonstrálják az „Egy kép felér ezer szóval" bölcsesség igazságtartalmát. A DOS használók által jól ismert (sok felhasználót visszatántorító) szöveges C:> prompt helyett a felhasználók a jóval barátságosabb, menükön keresztül vezérelhető ikonokat és programokat tartalmazó felülettel találják magukat szembe. A megjelenésnél is talán fontosabb hosszú távon a Microsoft Windows alkalmazások kézzelfoghatósága és felhasználóközelsége. Bármely Windows verzióról legyen is szó, az alkalmazások következetesek a felhasználói felületek használatában (legalábbis próbálnak következetesek lenni: a Windows 95 alkalmazások természetesen másként néznek ki, mint a Windows 3.X programok). A háttérben az az elgondolás áll,

hogy a felhasználónak több ideje marad az alkalmazások elsajátítására, és kevesebbet kell problémáznia azon, hogy melyik gomb lenyomása mit eredményez egy adott menün vagy dialógusablakon belül.

A programozók hosszú ideje vegyes érzelmekkel tekintenek a GUI-kre, de úgy tűnik, hogy kezdő felhasználók kedvelik őket. Ez volt az oka széles körű elterjedésüknek is. Ma már minden Windows alatt futó programtól elvárható, hogy GUI-alapokon működjön (továbbá rendelkezzen a kellő látvánnyal és kezelhetőséggel). Ezért, ha Windows alá kívánunk programokat fejleszteni, egy olyan eszközre van szükségünk, amelynek segítségével hatékonyan hozhatunk létre GUI-alapú alkalmazásokat.

Hosszú ideig nem volt ilyen segédeszköz. A Visual Basic megjelenése előtt Windows alkalmazások fejlesztése jóval nehezebb volt, mint DOS alá írni programot. A programozóknak túl sok mindenre kellett ügyelniük, többek között, hogy mit csinál az egér, a menün belül hol van pontosan a felhasználó, egyszer vagy kétszer kattintott-e az egérrel egy adott helyen. Windows alkalmazások fejlesztéséhez profi C programozónak kellett lenni, és többszáz sorból álló programkódot igényelt a legegyszerűbb feladat megvalósítása is. Még a szakemberek is gyakran gondban voltak. A Microsoft Software Development Kit (Microsoft Szoftverfejlesztő Csomag), amely nélkülözhetetlen segédeszköz volt a C fordító mellett, kb. 4 kg-ot nyomott.

A Visual Basic első három verziója részben megváltoztatta ezt a helyzetet, majd a Delphi 16 bites változata forradalmasította a Windows alkalmazások fejlesztését. Az új 32 bites verzió (közkeletű nevén Delphi 32, hivatalos nevén Delphi 2.0) a Windows 95 és Windows NT alá programozni kívánók számára nyújtja azokat a tulajdonságokat (sok új vonás megjelenése mellett), amelyeket a Delphi 16 nyújt a Windows 3.X alkalmazások fejlesztői számára. Igényes Windows alatt futó szoftverek készíthetők nagyságrendekkel rövidebb idő alatt, mint ahogy az korábban lehetséges volt. A forráskód méretének csökkentésével a hibák is ritkábban fordulnak elő, és ha előfordulnak, megtalálásuk és kijavításuk nem okoz akkora gondot. *A Delphi megjelenésével nemcsak hatékonnyá vált a programozás a Windows bármely verziójában, hanem szórakoztatóvá is* (a legtöbb esetben).

Függetlenül attól, hogy melyik Delphi változattal dolgozunk, a következő előnyök tűnnek ki a Visual Basic első négy változatával történő összehasonlításkor:

- A Delphiben fejlesztett alkalmazások lényegében ugyanolyan gyorsak, mintha C-ben vagy C++-ban dolgoztunk volna.
- A Delphi segítségével valódi végrehajtható programkódokat – köztük DLL-eket is (a DLL a *dynamic link library*-ból alkotott betűszó, a DLL-ek Windows programozás alapelemei) – hozhatunk létre.
- Újrahasználható objektumokat építhetünk az objektumorientált programozási modellnek megfelelően (l. 7. fejezet).

Természetesen a 32 bites változat a fentieket még jó néhány pozitív tulajdonsággal bővíti: például lehetőség nyílik lényegében korlátlan hosszúságú sztringek kezelésére a Visual Basic-ben megszokott egyszerűség megtartásával, és teljesen az OLE-t támogató alkalmazásokat pl. egy pénzügyi függvényeket tartalmazó csomagot hozhatunk létre. Ez a verzió egy új compilerrel is rendelkezik, amelynek üzenetei gyakran hasznos információkkal szolgálnak, ellentétben a Delphi 1.0 fordítójával.

Megjegyzés: *A Delphi 16 és a Delphi 32 kliens/szerver verzióiról nem esik szó a könyvben. Ezek megkönnyítik a lokális hálózatokra tervezett alkalmazások elkészítését.*

Hogyan fejleszthetünk Delphi alkalmazást?

Minden Delphi alkalmazás fejlesztésekor az első lépés annak a megtervezése, amit a felhasználó látni fog – azaz a képenyőablakok megszerkesztése. Milyen menüket szeretnénk? Milyen méretű ablakot használjon az alkalmazás? Hány ablakot kívánunk használni? Megengedjük a felhasználónak, hogy átméretezze az ablakokat? Hová helyezzük a nyomógombokat, amelyekre kattintva a felhasználó bizonyos funkciókat aktivizálhat? Lesz-e olyan hely a felületen (editor vagy memóablak), ahová a felhasználó szöveget írhat?

A Delphiben *komponenseknek* nevezik azokat az objektumokat, amelyeket a programozó a megszerkeszteni kívánt ablakokra helyezhet. Attól függően, hogy a Delphi melyik változatát használjuk, kezdetben több, mint 70 (a 16 bites verzió esetén) illetve több, mint 100 (a 32 bites verzió esetén) komponens áll rendelkezésünkre, amelyeket később kiegészíthetünk magunk készítette vagy vásárolt komponensekkel. A 4. feje-

zetben részletesen foglalkozunk majd a Delphihez mellékelt komponensekkel.

Megjegyzés: *Ha az olvasó rendelkezik bizonyos jártassággal az objektumorientált programozás (OOP) terén, akkor nem lepődik meg azon, hogy a komponensek gyakorlatilag OOP modellbeli objektumok. (A 7. fejezetben még többet tudhatunk meg az objektumorientált programozásról és arról, hogyan lehet a Delphi komponenseket beilleszteni az OOP fogalomkörébe.)*

A Delphi és a Visual Basic egy közös vonása a képernyő könnyű kezelhetősége. A programozó a szó szoros értelmében megrajzolhatja a felhasználói felületet, majdnem mint egy rajzprogram segítségével. Ezen felül a felületre helyezett parancsgombok és más komponensek, amelyeket egy kezdetben üres ablakra helyeztünk, automatikusan felismerik a felhasználó tevékenységeit pl. az egérmozgatást és az egérgomb kattintását. Ezeket a külső tevékenységeket eseményeknek nevezzük. A Delphi megkönnyíti a menük készítését is, néhány kézmozdulattal hozhatunk létre főmenü és legördülő menükomponenseket. A Visual Basic-kel ellentétben a Delphi megkönnyíti a korábbi alkalmazásokhoz készített felhasználói felületrészek újbóli felhasználását.

Csak a felület megtervezése után következik valami olyasmi, mint a hagyományos programozás. A Delphi-beli komponensek felismerik az olyan eseményeket, mint kattintás az egérrel, de az objektumok reakcióit ezekre a programozó határozza meg speciális eljárások írásával. Majdnem minden esetben szükség van programkód írására, ha azt akarjuk, hogy komponenseink válaszoljanak a külső behatásokra. Az itt leírt tulajdonságok különböztetik meg a Delphi alatti programozást a hagyományos eljárásorientált programozástól – pl. a Turbo Pascal korábbi változatai alatt történő fejlesztéstől.

A hagyományos nyelveken írt program felülről lefelé haladva hajtódik végre. Régebbi programnyelvek esetén a végrehajtás az első sortól indul és vonul végig a kódon, érintve a megfelelő részeket. Egy Delphi program általában teljesen másképp működik. A program magját olyan független kódok halmaza alkotja, amelyek kizárólag az általunk előzőleg kijelölt eseményre aktivizálódnak és válaszolnak. Ez alapvető változást jelent a program és a felhasználó viszonyában is. A programozó eddig úgy tervez-

te a programot, hogy az teljesen alá legyen rendelve elképzeléseinek. Így viszont a felhasználónak is nagy szerep jut.

A Delphi-beli kódok, amelyek meghatározzák, hogy egy adott program hogyan reagáljon olyan eseményekre, mint pl. a kattintás az egérrel, az ún. eseménykezelő eljárásokban (vagy rövidebben eseménykezelőkben) gyökereznek. Az eseménykezelő eljárás nem más, mint egy olyan programtörzs, amely csak egy meghatározott külső esemény bekövetkezésének hatására hajtódik végre. Majdnem minden esetben a Delphi program minden végrehajtható részlete vagy maga egy eseménykezelő eljárás, vagy egy eseménykezelő eljárás használja feladatának végrehajtásához. Valójában azért, hogy az elnevezésekben is hangsúlyt kapjon a Delphi különbözősége a többi programnyelvtől, a dokumentáció a *program* kifejezés helyett a *projekt* (project) szót használja, utalva arra, hogy a programkód és a felhasználói felület kombinációja alkotja a Delphi alkalmazást.

A következő összefoglalás mutatja, hogy milyen lépcsőkön keresztül történik a Delphi alkalmazások megtervezése:

1. A felhasználó által látott ablakok testre szabása.
2. Annak eldöntése, hogy milyen eseményekre reagáljanak az adott ablakban található komponensek.
3. Az eseménykezelő eljárások (és az ezeket kiszolgáló másodlagos eljárások és objektumok) megírása a fenti eseményekhez.

Az alkalmazás futtatásakor a alábbiak történnek a színfalak mögött:

1. A Delphi megvizsgálja az összes ablakot és az azon lévő összes komponenst, hogy történt-e valamilyen esemény, amelyet az adott vezérlőelem felismer (egérmozgatás, kattintások, billentyűleütések stb).
2. Ha a Delphi észlel egy eseményt, amelyhez nincs beépített automatikus válasz, akkor megvizsgálja, hogy van-e az alkalmazásban az eseményhez rendelt eseménykezelő eljárás.
3. Ha van, akkor a Delphi végrehajtja azt, és újra az 1. pont lép érvénybe.
4. Amennyiben nem talál ilyet, akkor vár a következő eseményre az 1. pontnak megfelelően.

Ezek a lépések ciklikusan követik egymást, amíg az alkalmazás fut. Általában minden változást valamilyen esemény előz meg. Ezért az eseményvezérelt programok inkább *reaktívak* mint *aktívak*, ami miatt jobban rászolgálnak a *felhasználóbarát* jelzőre.

Mivel a Delphi programnyelve az Object Pascalon alapul, egyszerűen készíthetünk nagy programokat a modern moduláris és objektum-orientált technikákra támaszkodva. A Delphi fegyvertárába tartozik egy hatékony nyomkereső program, amellyel minimálisra csökkenthetjük a végtermék hibáit, és azon keresztül annak az esélyét, hogy az alkalmazás futás közben valamilyen módon működésképtelenné válik. A Delphi fordítója rendkívül gyors, ami azt jelenti, hogy az alkalmazás fejlesztésének első szakaszában oly gyakori rutin- és gépelési hibákat pillanatok alatt ki lehet javítani. Szót érdemel még a kiterjedt beépített help-rendszer is, amely a gyors információszerzést biztosítja fejlesztési időben.

A Delphi működtetési feltételei

A Delphi egy bonyolult program. Ha a 16 bites változattal kívánunk dolgozni, biztosítanunk kell a következő feltételeket:

- Legalább 386-os CPU 6 MB RAM-mal. (Valójában az optimális működés biztosításához egy gyors 486-os vagy Pentium és legalább 8 MB RAM szükséges.)
- Legalább 27 MB szabad terület a winchesteren. (A teljes kliens/szerver verzióhoz legalább 90 MB kell!)
- Egér, vagy egérrel kompatíbilis pozicionáló eszköz.
- Enhanced módban futó Microsoft Windows 3.X vagy újabb verzió.

Megjegyzés: Habár a Delphi 16 (Delphi 1.0) működik Windows NT és Windows 95 alatt is, ez csak 16 bites alkalmazásként fut, és kizárólag 16 bites alkalmazásokat lehet vele létrehozni.

A 32 bites verzió futtatásához a következő feltételek szükségesek:

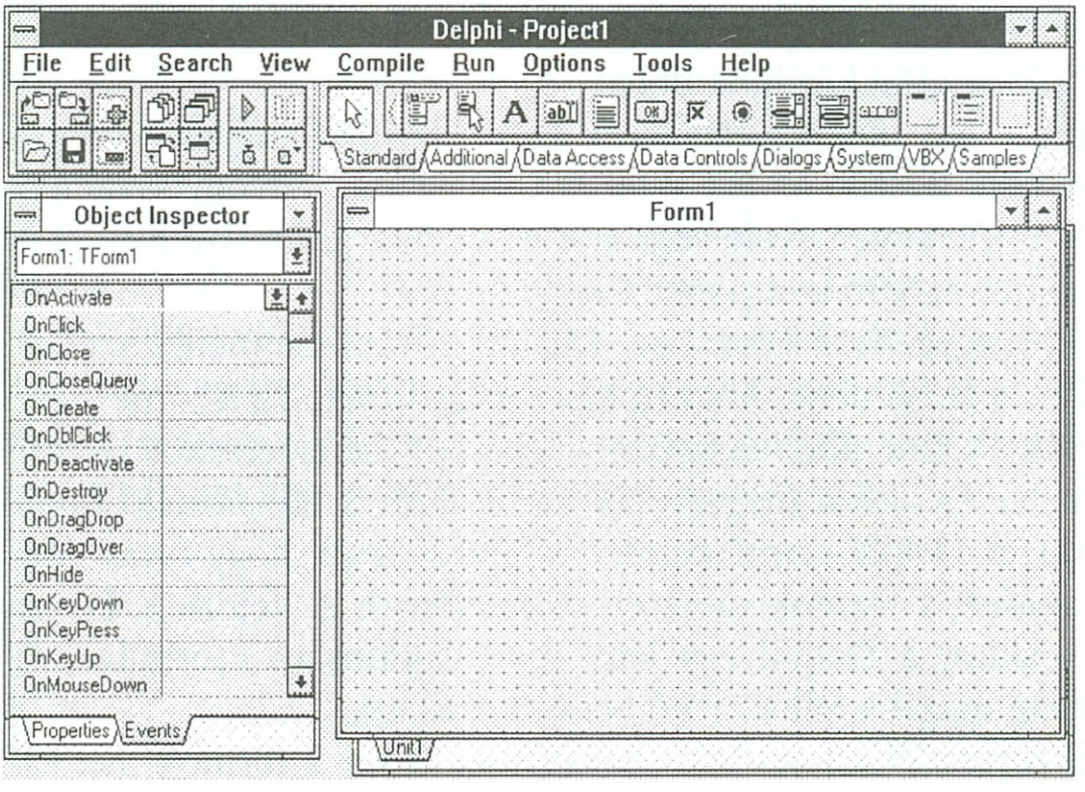
- Microsoft Windows 95 vagy Windows NT 3.51 vagy újabb verzió.
- Legalább egy gyors 486-os vagy Pentium és minimum 16 MB RAM.
- Legalább 40 MB szabad terület a winchesteren. (A teljes kliens/szerver verzióhoz legalább 90 MB kell!)
- Egér vagy egérrel kompatíbilis pozicionáló eszköz.

■ *Hogyan dolgozzunk a Delphivel?*

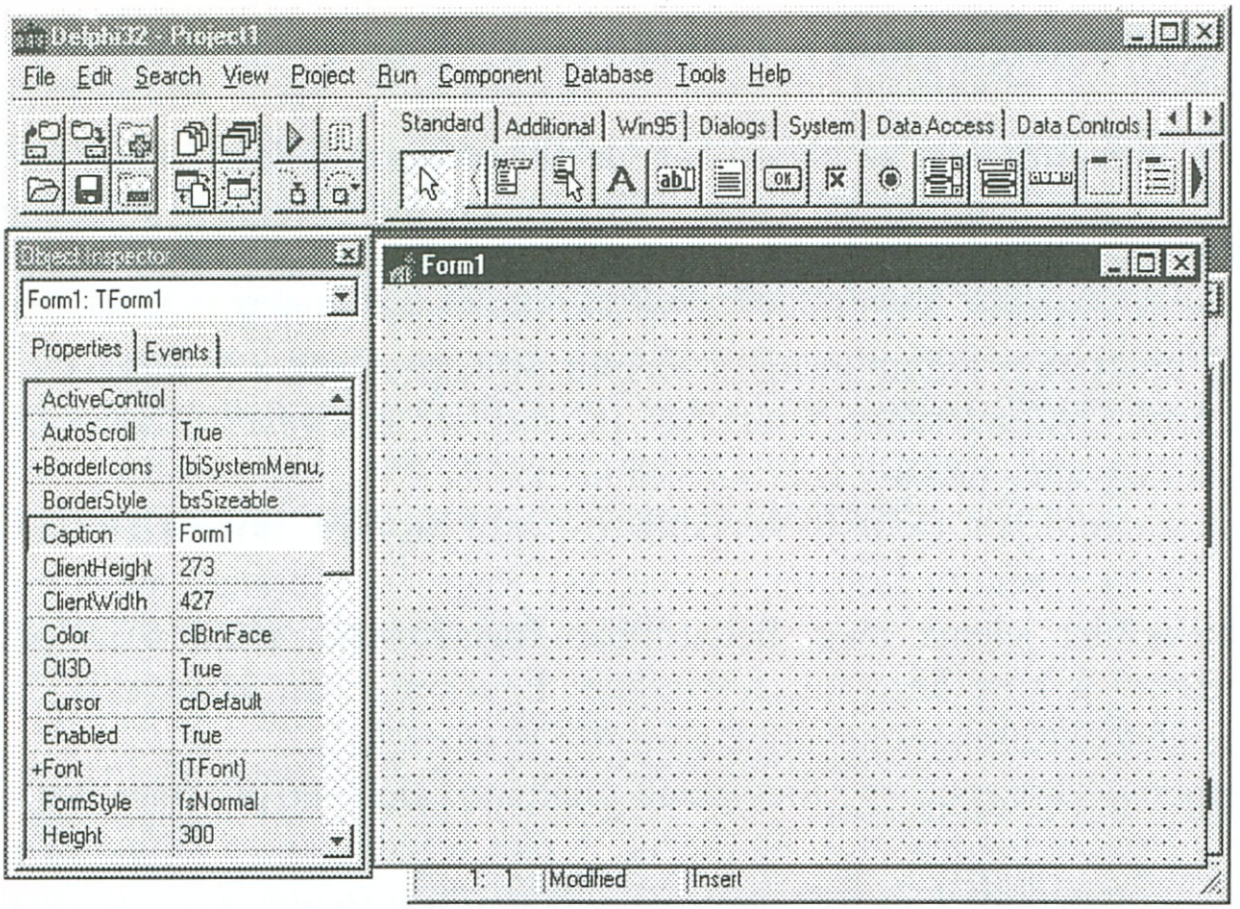
Ebben a fejezetrészben az olvasó két Delphi program fejlesztésének menetét követheti nyomon. A lépések lényegében megegyeznek a Delphi mindkét verziójában. Az első program a hagyományos „Helló Világ" program módosítása, a második pedig egy bittérkép-megjelenítő. A megjelenítő különösen kidomborítja a Delphi ütőképességét a Windows alkalmazások fejlesztésének terén: két programsor az egész. C-ben vagy C++-ban ugyanez néhány száz sort jelentene.

Amikor elindítja a Delphit, kezdetben – attól függően, hogy a 16 bites vagy a 32 bites verziót használja-e – az 1-1. vagy 1-2. ábrának megfelelő képernyőfelület jelenik meg. Behatóbban foglalkozunk majd a Delphi környezettel a következő két fejezetben. Egyelőre talán könnyebb, ha arról megfeledkezünk, és követjük a megfelelő lépéseket.

A képernyő közepén egy üres ablak található, amelyet pontrács borít. Ebben a *formban* fogunk most építkezni. A rács segédeszköz a komponensek (pl. nyomógombok és listaablakok) képernyőn történő elhelyezéséhez (l. részletesebben a 4. fejezetben). Amikor egy projektet lefuttatunk (vagy lefordítunk, amivel lehetővé válik a Delphi környezettől független futtatása), a formok lesznek a felhasználó által látható ablakok.



1-1. ábra. A Delphi 16 bites verziójának képernyője a bejelentkezés után



1-2. ábra. A Delphi 32 bites verziójának képernyője a bejelentkezés után

Az üres form tetején a az ablak nevét (caption) tartalmazó címsor látható. Jelenleg a Form1 szöveg szerepel a címsorban az alapbeállításnak megfelelően, a Delphi mindig ezt a címet jeleníti meg a formban, valahányszor egy új projekttel kezdünk dolgozni.

A Form1 ablak bal oldalán található az objektum-felügyelő (Object Inspector). (Ha nem látszik, az F11 vagy ALT-V-O gomb lenyomásával előhozhatjuk.) Az objektumfelügyelő segítségével szabhatjuk testre formjainkat és az azokra helyezhető komponenseket.

Koncentráljunk egyelőre a Form1-re. Mielőtt továbbmennénk, szükséges, hogy a felhasználó ismerje, hogyan lehet megváltoztatni a form méretét és helyét. Sok Delphi alkalmazásban a fejlesztés befejezésekor látható form mérete, helye és alakja megegyezik azzal, amit a felhasználó futási időben tapasztal. A Delphi természetesen lehetővé teszi, hogy futási időben is változtassunk a formok méretén és helyén (l. 4. fejezet). Ha ez nem így lenne, akkor a Delphi egy lényeges tulajdonsága csorbulna, ti. hogy dinamikusan reagál a felhasználó beavatkozásaira.

Az egyik Microsoft Windows-ban szokásos módszer a formok méretezésére a következő: először rákattintunk az ablakra, ezzel azt aktívvá

tesszük, majd az egér mozgatásával a nyilat (kurzort) a form egyik szélére vagy sarkára állítjuk, ahol a sima nyíl kétfejű nyíllá változik. Ebben a helyzetben, ha a bal oldali egérgombot lenyomva tarjuk, és az egeret elkezdjük mozgatni, az ablak méretete változik. Az egérgomb felengedése után a méret rögzül. Hasonlóképpen történik a form mozgatása is: a címsorra kattintunk, az egérgombot lenyomva tartjuk, majd a kívánt helyre húzzuk az ablakot. A form mérete és helye a form *tulajdonságainak (properties)* egy-egy eleme.

A legelső fejlesztésünk első lépései a következők:

1. Változtassunk az form alapértelmezett méretén, alakján és helyén – a fent leírt módon – az oldalak kihúzásával és az ablak arrébb vitelével.
2. Indítsuk el a projektet a Run menü Run pontjának választásával.

Vegyük észre, hogy amit kaptunk, nem más, mint egy teljesen megszokott ablak, amelynek helye, alakja és mérete a fejlesztési időben beállítottakkal egyezik meg.

Figyelemre méltó még, hogy a futtatáskor megjelenő ablak a Windows ablakok megszokott tulajdonságaival rendelkezik, úgymint méretezhető keret, rendszermenü (a bal felső sarokban), maximalizáló és minimalizáló gomb (a jobb felső sarokban). (Windows 95 alatt a jobb felső sarokban még a kilépés [exit] gomb is megjelenik.) Ezek a Delphi legfontosabb vonásait illusztrálják: anélkül, hogy bármit is kellett volna tennünk, az alkalmazásaink automatikusan a Windows adott verziójához idomulnak, és az ottani szabványoknak megfelelően viselkednek.

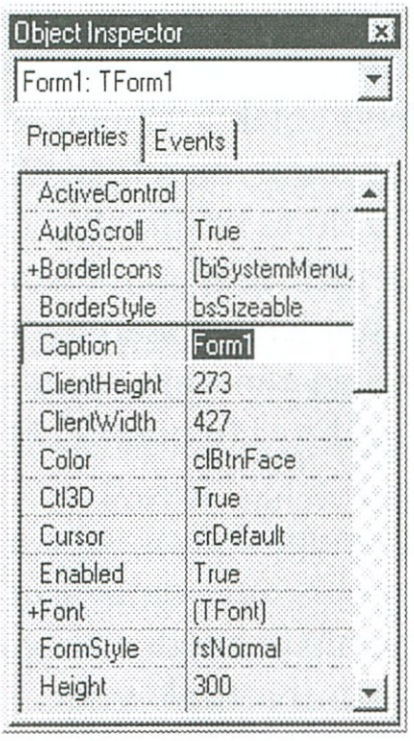
Térjünk vissza a fejlesztői környezetbe az ALT-F4 lenyomásával vagy a kontrol box-ra történő kettős kattintással. Az alkalmazás automatikusan „tudja", hogyan kell reagálnia az ablakzárás konvencionális módszereire. (A 32 bites változat esetén a kilépés gombra is kattinthattunk volna.) Nem lehet eleget hangsúlyozni, hogy szinte semmit sem kell a programozónak tennie annak érdekében, hogy a Delphi alkalmazás úgy viselkedjen, ahogy azt egy Windows felhasználó elvárja.

Egy „Helló Világ" típusú program

Ebben a részben láthatunk majd egy olyan példa-projektet, amely egy üres ablak címsoraként megjeleníti – válaszként az egér kattintására – a „Helló új felhasználó" szöveget. Egy kicsikét azonban komplikáljuk a feladatot azzal, hogy kezdetben az ablak címsorába az alábbi szöveget írjuk: „Az egér kattintásával indul!"

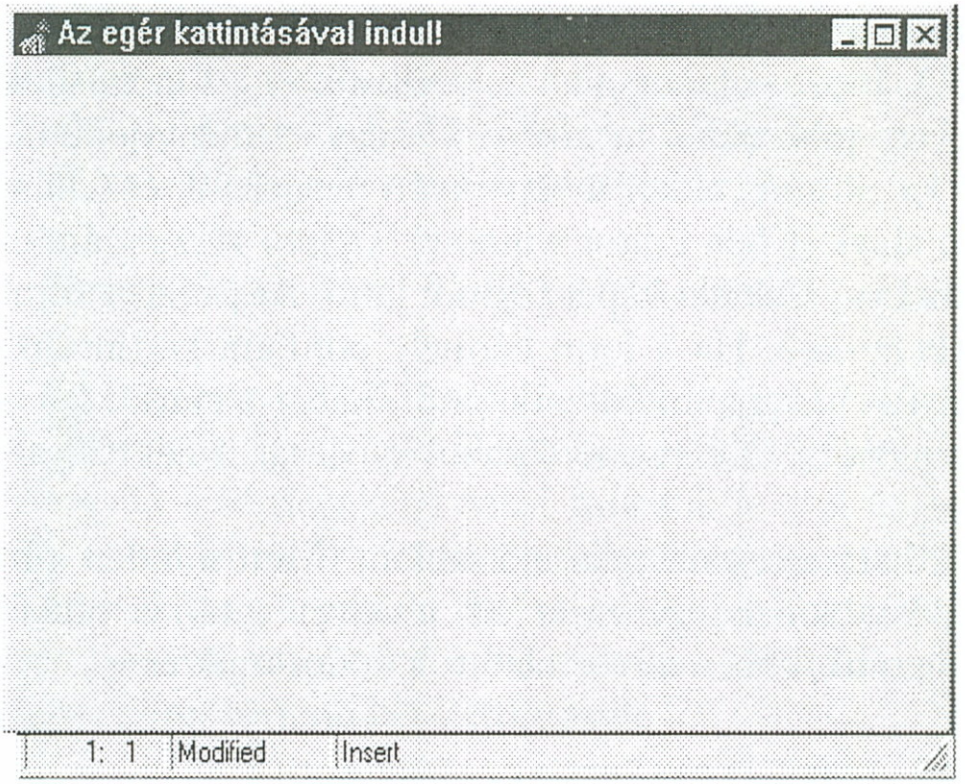
Az ezt megvalósító szükséges lépések (a Delphi mindkét változtában működnek):

1. Az objektumfelügyelőben (Object Inspector) válasszuk a *properties* (tulajdonságok) oldalon a „Caption" pontot, az alábbi ábrának megfelelően. (Hasonlóan néz ki a Delphi mindkét változatában.) A „Form1" felirat kiemelve jelenik meg.



2. Írjuk be „Az egér kattintásával indul" szöveget az objektumfelügyelő Caption sorának jobb oldali cellájába.

Ezzel a formhoz tartozó tulajdonságokat (properties) állítottuk be. Ha az F9 lenyomásával vagy a Run | Run pont választásával lefuttatjuk a (fejlesztés alatt álló) projektet, Windows 95 alatt az 1-3. ábrán látható kép jelenik meg a képernyőn. (A Windows 3.X verziója alatt is hasonló felületet kapunk néhány tipikus Windows 95-ös elem nélkül, mint pl. a kilépés nyomógomb.)



1-3. ábra. A „Helló új felhasználó" projekt első lépése

Megjegyzés: *Elképzelhető, hogy az általunk használt változat felkínálja munkánk elmentését a fordítás megkezdése előtt. Ebben az esetben nyugodtan fogadjuk el a Delphi által javasolt neveket.*

Vegyük észre, hogy ez az ablak meglehetősen érzéketlen, sem a szimpla sem a kettős kattintásra nem reagál. Vegyük észre, hogy a standard ALT-F4 billentyűkombináció megfelelően működik – használatával a project futása befejeződik, és visszakerülünk a fejlesztői környezetbe.

Mivel minden Microsoft Windows program (és ezért a Delphi projekteké is) lényege az, hogy a formokkal végzett felhasználói műveletekre válaszol, változtassunk a projektünkön úgy, hogy a címsorszöveg kicserélődjön, ha az egérrel a formra kattintunk. Ebben az esetben – ahogy mindjárt látni fogjuk – egyetlen programsor is elegendő. Általában a válaszreakciókat megvalósító programrészek mérete az adott feladattól függ.

Az első eseménykezelő eljárás

Az eseménykezelő eljárások használatának illusztrálása előtt győződjünk meg arról, hogy az előző progamot megállítottuk (pl az ALT-F4 lenyomásával). Majd kattintsunk az Events fülre az objektumfelügyelőben. (Az

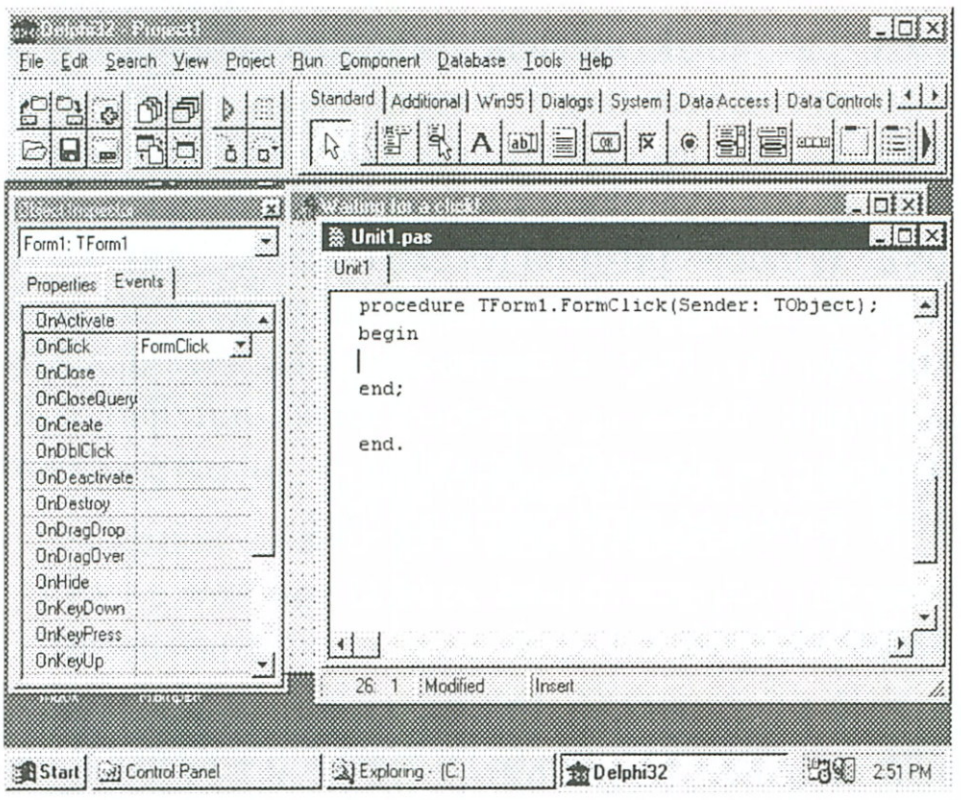
1-1. és az 1-2. ábrák alapán láthatjuk, hogy Windows 95 alatt az Events fül felül Windows 3.X használatakor alul helyezkedik el.) Kattintsunk kétszer az „OnClick" sor jobb oldali mezőjére. Ha egy esemény jobb oldali cellájába kétszer kattintunk, akkor a képernyőn megjelenik és felülre kerül a kódszerkesztő ablak (Code Editor window). Delphi verziószámtól függetlenül az 1-4. ábrához hasonló képet fogunk látni. Az ábrán a form alulra került, de így is látható. Ha a form bármely pontjára kattintunk, akkor az felülre kerül, és eltakarja a kódszerkesztőt, habár a címsor így is jelzi jelenlétét a form mögött. (A címsorra történő kattintás ismét előhozza a kódszerkesztőt.)

Az 1-4. ábrán jól látható az eseménykezelő sablon. (Mint minden sablon, az eseménykezelő sablon is egy vázat ad, amelyet aztán a célnak megfelelően kiegészíthetünk, ebben az esetben itt a kódot írjuk meg.)

```
procedure TForm1.FormClick(Sender: TObject);
begin

end;

end.
```



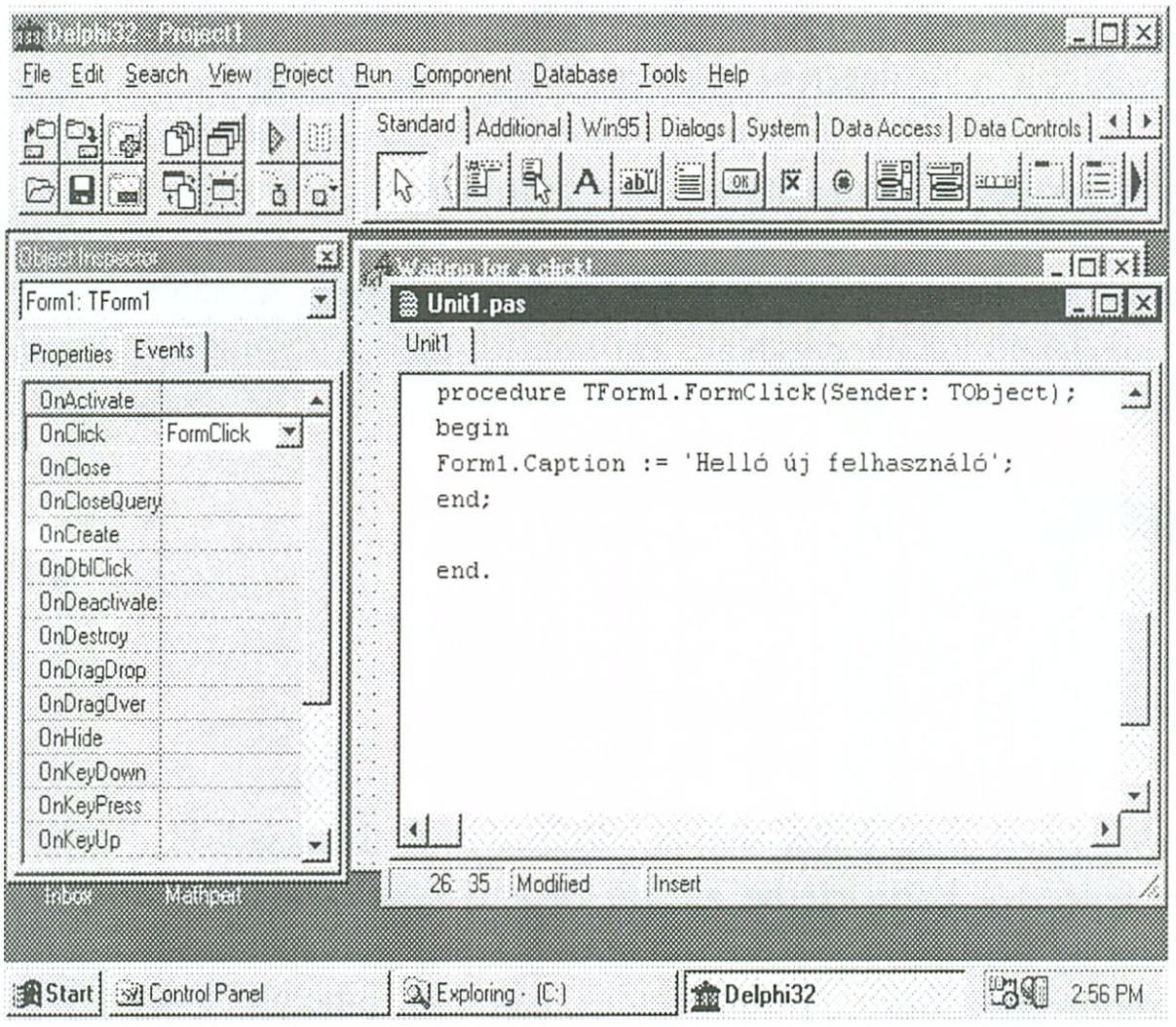
1-4. ábra. Az eseménykezelő váz ablaka

Lehet, hogy egy kicsit még homályos az eseménykezelő eljárásban használt szintaxis, de jelenleg ne aggódjunk emiatt. Az 5. és 7. fejezetben ezzel is megismerkedhetünk. Állítsuk a kurzort a „begin" szó utáni szabad sorra, és írjuk be a következő utasítássort:

Form1.Caption: = ´Helló új felhasználó´;

A képernyőn az 1-5. ábrához hasonló képet fogunk látni. (Visual Basic programozók figyelem: a Delphiben az idézőjelek helyett aposztrófok közé tegyük a sztringeket!) Most futassuk le a projektet (az F9 lenyomásával), és kattintsunk a formra: a felirat megváltozik.

Állítsuk meg a projekt futását, és térjünk vissza a Delphibe az ALT-F4 lenyomásával vagy a fejléc bal sarkában található „gyufásdobozra" (rendszermenüre) történő két kattintással.



1-5. ábra. A befejezett első eseménykezelő eljárás kódja

Egy komponens hozzáadása

Alkalmazásunk eddig nem tartalmazott komponenst. Változtassunk a projektünkön egy parancs nyomógomb beikatatásával, amelyre kattintva a felirat kicserélődik. Ugyanakkor érjük el azt, hogy ha a nyomógombon kívülre kattintunk, megjelenjen egy üzenetablak, amely arra szólítja fel a felhasználót, hogy kattintson a gombra. Ehhez kövessük az alábbi lépéseket:

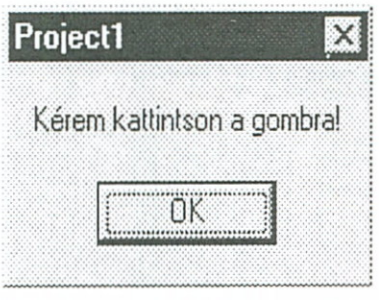
1. Állítsuk a kurzort arra a sorra, amelyet begépeltünk, és válasszuk ki azt a szokásos Windows technikák egyikével (egér húzásával, miközben a bal oldali gombot lenyomva tartjuk, vagy a SHIFT NYÍL segítségével).

(Ha a form látható, de a kód nem, kattintsunk először a kódszerkesztő látható részére vagy nyomjuk meg az F12-t.)

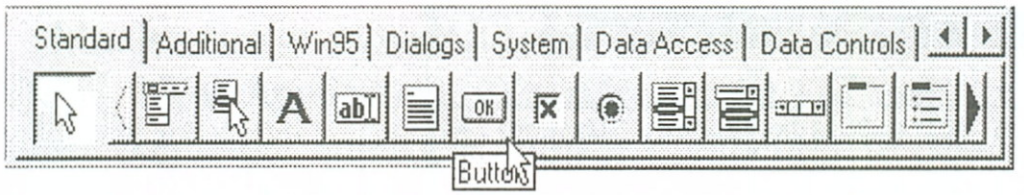
2. Cseréljük ki a kiválasztott kódsort a következő szövegre:

ShowMessage(´Kérem kattintson a gombra!´);

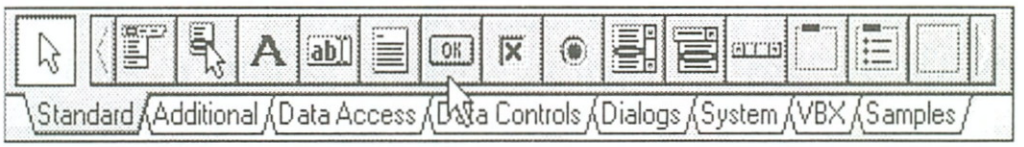
Ha most elindítjuk a programot, akkor fel fog tűnni az alább közölt képnek megfelelő kis üzenetablak. Ha rákattintunk a nyomógombra, az ablak eltűnik.



Természetesen még hátra van a parancsgomb hozzáadása. Térjünk vissza a fejlesztői környezetbe az ALT-F4 lenyomásával. (Kattintsunk a formra, ha az nem látható.) Kattintsunk a komponenspalettán (l. ábra) lévő nyomógomb (Button) komponens ikonjára, amelyre a következő ábrán a kurzor mutat.

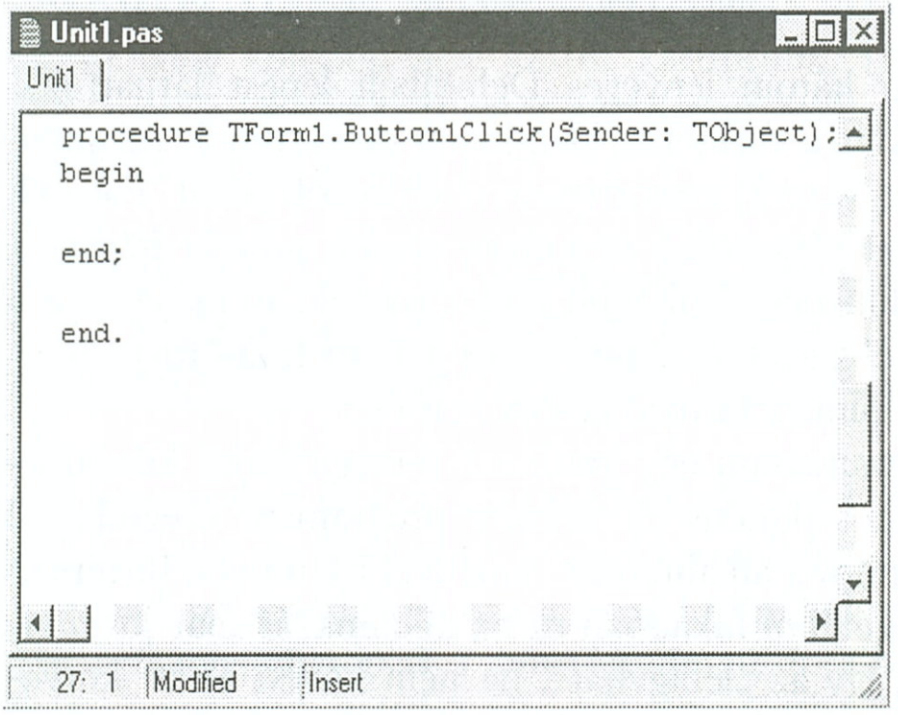


Windows 3.X alatt a komponenspaletta így néz ki:

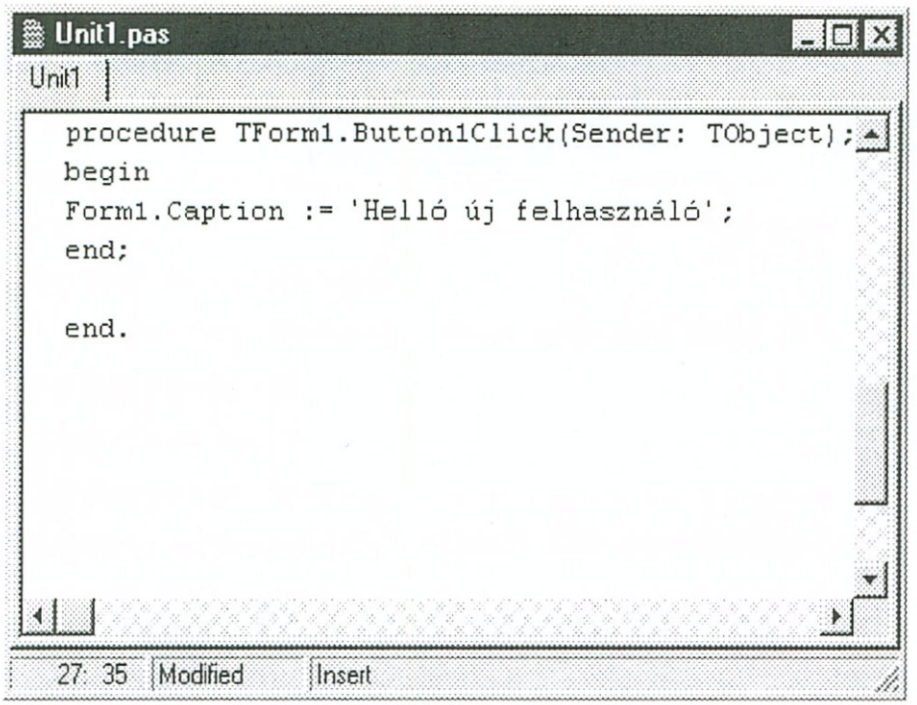


(Mindkét esetben a nyomógomb ikonja ugyanaz, és a komponenspaletta Standard oldalán található. Ez is jól mutatja, hogy a 32 és a 16 bites verziók sokban megegyeznek.)

Így a képernyő közepére helyeztünk egy nyomógombot az alapértelmezett alakkal és mérettel. (A 4. fejezetben látni fogjuk, hogyan lehet a komponensek méretét és alakját megváltoztatni, és többet tudhatunk majd meg a parancsgomb tulajdonságairól is.) Ha kettőt kattintunk a parancsgombra, megjelenik egy eseménykezelő sablon az OnClick esemény nyomógombos (button) változatához, miként az itt látható:



Írjuk be a **Form1.Caption:= ´Helló új felhasználó´;** sort a begin szó után.



A projektet lefuttatva láthatjuk, hogy az a leírtaknak megfelelően viselkedik. Ha a nyomógombra kattintunk a felirat megváltozik, bárhová máshová történő kattintáskor pedig megjelenik az üzenetablak.

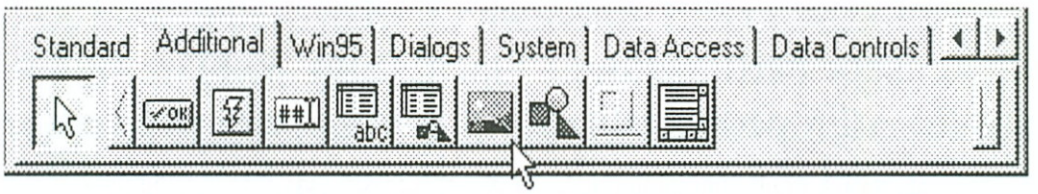
■ *Egy nagyobb projekt készítése*

Az előző fejezetrészben három lényeges Delphibeli lépést láthattunk: komponensek hozzáadását a formhoz, Delphibeli objektumok tulajdonságainak beállítását és (egyszerű) eseménykezelő eljárások írását. Fogjunk most neki egy sokkal valóságszerűbb és kifinomultabb projekt írásának, amelyhez meglepő módon nem kell több kódot írnunk. Remélem, ez a projekt meghozza majd étvágyát a Delphihez, és demonstrálni fogja, hogy milyen ütőképesek a Delphihez tartozó komponensek.

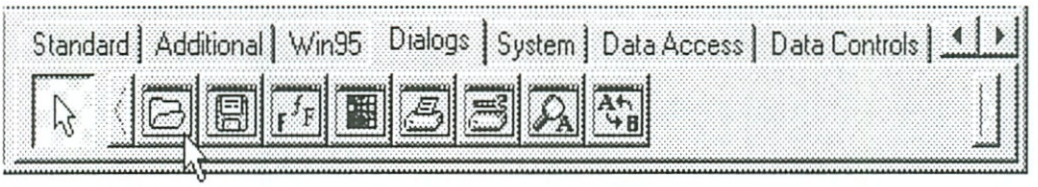
A célom az, hogy bemutassam egy grafikusfájl-megjelenítő készítésének lépéseit. Néhány komponens és néhány programsor megírásával létrehozhatunk egy Windows alkalmazást, amellyel bármilyen bittérkép megjeleníthető. Természetéből fakadóan ez az alkalmazás már jóval érzékenyebb mint elődje, így az „lefagyhat", ha nem követi az itt körvonalazott lépéseket pontosan. Jelenleg még nem tudhatjuk, hogyan tegyük alkalmazásainkat „bombabiztossá", hogyan mentsük el azokat, és hogyan készítsünk valóban felhasználóbarát projekteket (Ne felejtsük el, ez csak az első fejezet!).

Mindenekelőtt nyissunk meg egy új projektet. A Delphi 32-ben ezt a File | New Application választással tehetjük meg, miközben minden olyan kérdésre, amely a munkánk elmentésére irányul nemmel (No gomb) válaszolunk. Ha a 16 bites változattal dolgozunk, válasszuk a File | New menüpontot, és most is a No gomb megnyomásával válaszoljunk minden olyan kérdésre, amely a projekt elmentését érinti.

Ezúttal két komponenst adunk az üres formhoz: egy *image* (kép) komponenst a grafika megjelenítéshez, és egy *OpenDialog* komponenst a könyvtár- és fájlstruktúra kezeléséhez. Függetlenül attól, hogy a Delphi melyik változatát használjuk, a képkomponenst úgy vihetjük a formra, ha először a komponenspaletta Additional feliratú fülére egyet, majd az Image komponens ikonjára (amely úgy néz ki, mint egy tájkép) kettőt kattintunk. Az ikont az ábrán a kurzor hegye mutatja:

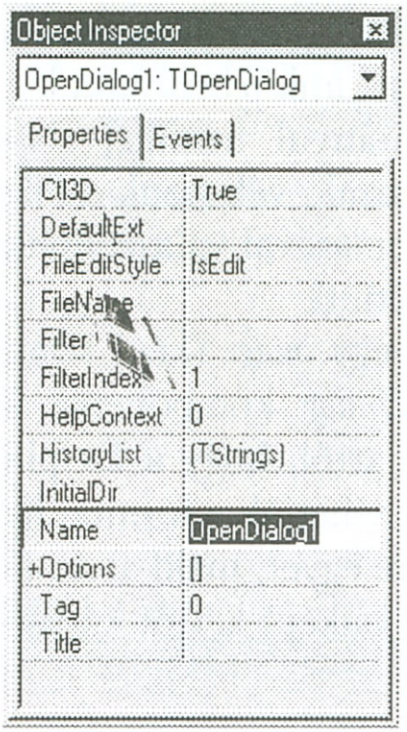


Ezután adjuk az OpenDialog-ot a formhoz úgy, hogy először a komponenspaletta Dialogs feliratú fülére egyet, majd az ábrán is látható OpenDialog komponensre kettőt kattintunk. (Ismét teljesen mindegy, hogy melyik Delphi változatot használjuk, ezek a standard komponensek mindig elérhetők lesznek.)

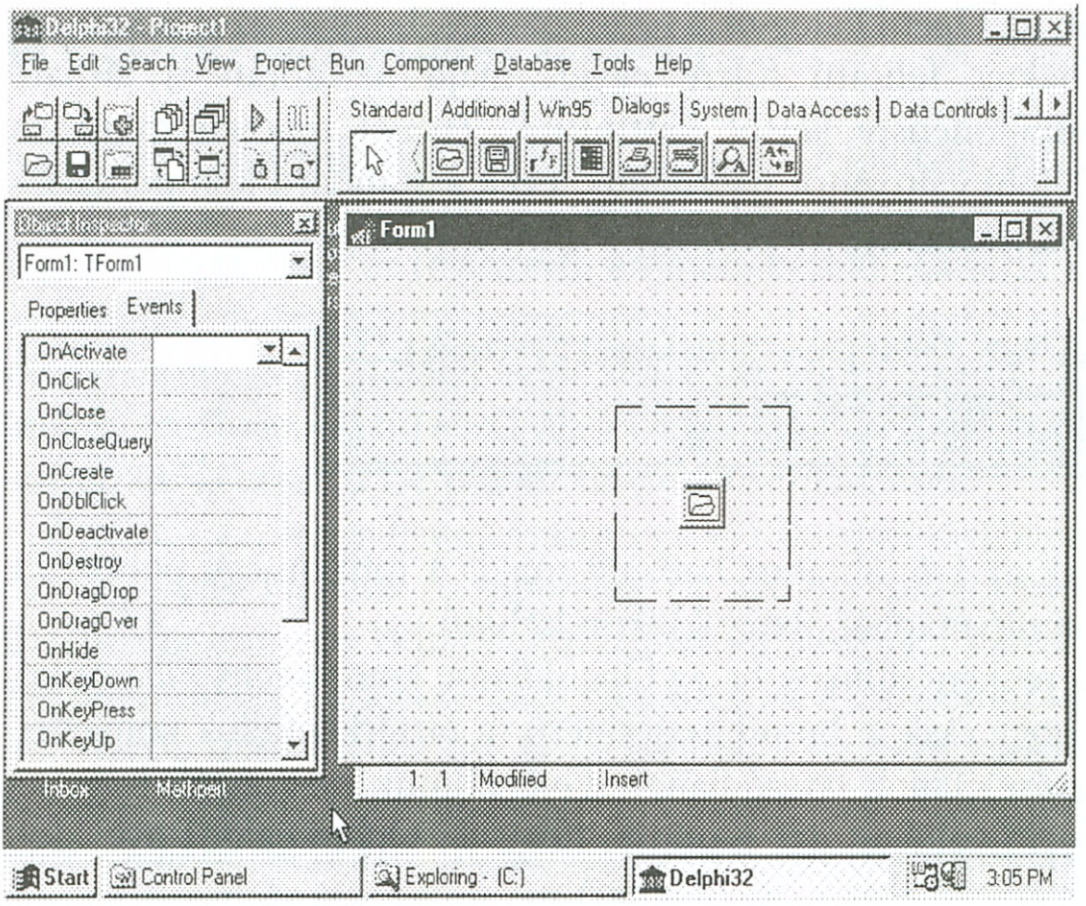


A képernyőnk az 1-6. ábrához hasonlít majd. A formon egy pontozott keretben lévő, nyíló mappát ábrázoló ikon látható.

Következő lépésként állítsuk be az OpenDialog komponens *Filter* tulajdonságát úgy, hogy csak a .bmp kiterjesztésű (a bittérképfájlok konvencionális kiterjesztése) fájlok látszódjanak. Győződjünk meg arról, hogy az objektumfelügyelő valóban azt mutatja, hogy az OpenDialog1-gyel dolgozunk (l. a következő ábra a 36. oldalon).



Ha nem ezt látjuk, akkor kattintsunk az OpenDialog ikonjára (l. 1-6. ábra), majd kattintsunk a Properties feliratú fülre, ekkor már valóban az OpenDialog1 komponenssel dolgozunk. Keressük meg a Filter tulajdon-



1-6. ábra. A bittérkép-megjelenítő kiindulási képernyője

ságot az objektumfelügyelőben, majd kattintsunk a jobb oldali cellára és írjuk be: ***.bmp**. Ez biztosítja, hogy az OpenDialog ablak csak a bmp (konvencionális bittérkép kiterjesztés) kiterjesztésű fájlokat jeleníti meg. Ezután írjuk be a Title (cím) tulajdonság sorának jobb oldali cellájába a **„Válassza ki a kívánt bittérkép fájlt!"** szöveget.

Egyébként ha most próbálnánk lefuttatni a projektet, akkor semmi sem történne. Az OpenDialog ablakot aktivizálni kell ahhoz, hogy megjelenjen. Ehhez egy új Delphi eszközt kell igénybe vennünk: a *metódust (method)*. Nagyjából arról van szó, hogy a tulajdonságok meghatározzák, hogy a komponensek milyenek legyenek, míg a metódusok azt mondják meg, hogy mit csináljanak. Az események pedig általában a felhasználó által kiváltott beavatkozások, amelyek bekövetkezése valamilyen kódrészlet végrehajtásához vezet.

A most felhasználandó metódust Execute-nak hívják, ezzel kell kibővítenünk a form OnActivate eseménykezelő eljárásának a megjelenítést végző programsorát. Az OnActivate eseménykezelő eljárás igen sajátos abból a szempontból, hogy a Delphi ezt minden alkalommal meghívja, amikor egy form aktívvá válik (ami a projektek indításakor nyilvánvalóan megtörténik).

Az OnActivate eseménykezelő sablon eléréséhez a következőket kell csinálnunk:

1. Kattintsunk a form bármely nem használt részére, így tudjuk az objektumfelügyelőben a formot kiválasztani (későbbi fejezetekben más módszerekkel is megismerkedünk).
2. Kattintsunk az objektumfelügyelő Events feliratú fülére.
3. Kattitsunk kettőt az OnActivate esemény sorának jobb oldali cellájára.

Ha minden igaz, a fentiek által megnyitottuk az OnActivate eseményhez tartozó eseménykezelő sablont. Most gépeljük be a következő két programsort, miután az eljárás a következőképpen fog kinézni:

```
Procedure TForm1.Activate(Sender: TObject);
begin
  OpenDialog1.Execute;
  Image1.Picture.LoadFromFile(OpenDialog1.FileName);
end;
end.
```

Ennyi. Ha most lefuttatjuk a projektet, akkor egy megszokott Windows-os dialógusablak bukkan fel („Válassza ki a kívánt bittérképfájlt!" felirattal), amelyben kizárólag .bmp kiterjesztésű állományok jelennek meg. Mozoghatunk meghajtóink könyvtárai között az egérrel történő kattintások segítségével, ahogy azt a Windows Open dialógusablakainál megszokhattuk. Keressünk egy bittérképet, kattintsunk rá, és a Delphi megjeleníti a formon.

Tipp: *A Delphi könyvtárának IMAGES alkönyvtárában sok bittérképet találhatunk.*

Nyilvánvaló, hogy ez az alkalmazás nem „bombabiztos": ha megnyomjuk az OK-t anélkül, hogy egy bittérképet is kiválasztottunk volna (vagy rákattintunk a Cancel gombra) az alkalmazásunk „lefagy". (Ha véletlenül ez történne, zárjuk be az összes üzenetablakot, és nyomjunk ALT-F4-et vagy válasszuk a Run | Program Reset menüpontot, hogy visszakerüljünk a fejlesztői környezetbe.)

Megjegyzés: *Az alkalmazás bombabiztossá tételéhez sem kell néhány kiegészítő sornál több. A 13. fejezetben ezt is látni fogjuk.*

Törékenysége ellenére ez az alkalmazás sok olyan technikát mutat be, amelyeket később újra meg újra felhasználhatunk a Delphi alkalmazások írásakor. A felület és a programkód bonyolultabbá fog válni, de a lényeg nem változik. Megtanulhattuk, hogy

- a komponenseket tulajdonságaik megváltoztatásával szabhatjuk testre;
- viselkedésükért a metódusok felelősek.

Azt is tapasztalhattuk, hogy a Delphi komponenseinek sok beépített funkciója van. Ez még nyilvánvalóbb lesz, ha majd a többi komponenst is használni fogjuk.

2. FEJEZET

A Delphi 32 programozói környezet

Ebben a fejezetben az olvasó megismerkedhet a Delphi 32 (Delphi 2.0) környezetét alkotó menük és ablakok kezelésével. (A fejezetben a „Delphi" szón mindig Delphi 32-t fogunk érteni, és a „Windows" mindvégig Windows 95-öt fog jelenteni, habár a Delphi 32 fut Windows NT 3.51 vagy újabb verziók alatt is.) Ha nem mozgunk teljesen otthonosan a Microsoft Windows 95 alkalmazások világában, akkor ez a fejezet bizonyára segíteni fog. Végső soron a Delphi maga is egy jól megírt Windows 95 alkalmazás, így annak minden építőköve a Windows 95 programokra jellemzően viselkedik. Tapasztalt Windows felhasználók lehet, hogy a fejezet egyes részeinek kihagyása mellett döntenek. Fontos, hogy amíg nem vagyunk teljesen tisztában a Windows alkalmazásokkal, addig nem leszünk képesek a Delphi lehetőségeit teljes mértékben kihasználni. A fejezet végére rutint szerzünk a Delphi beépített súgójának, fájlkezelőjének és szerkesztési kellékeinek használatában. Ha személyes használatra készítünk alkalmazást a Delphiben, akkor nem szükséges megfelelnünk a Windows szabványoknak. Azonban, ha az alkalmazást mások is használják, a szabványok követése megkönnyíti kezelésének elsajátítását. Nekünk is ügyelnünk kell arra, hogy betartsunk olyan egyszerű dolgokat, amelyek már beleivódtak a Windows felhasználók vérébe, pl. hogy az egyszeres kattintással kiválaszthatnak egy menüpontot, kettős klikkelésssel aktivizálhatnak egy adott alpontot.

Megjegyzés: A Windows 95 alkalmazások összegyűjtött irányelvei The Windows Interface Guidelines for Software Design (ISBN 1-55615-679-0) című Microsoft-kiadványban található meg. Érdeklődjön a Microsoftnál vagy a könyvesboltokban az ár és az elérhetőség iránt.

A Microsoft Windows irányelveknek megfelelő beállításokat a Delphi alapértelmezésben tartalmazza. Miként azt már az előző fejezetben is láthattuk, a komponensek a szokásos alakkal és építőelemekkel rendelkeznek, mozgathatók és méretezhetők, ahogy azt a felhasználó el is várja. Hasonlóképpen a menük is a megszokott módon reagálnak a külső behatásokra. Természetesen a Delphi nem erőlteti ránk ezeket az alapértelmezéseket, de tény, hogy nem túl jó ötlet azokon meggondolatlanul és ok nélkül változtatni.

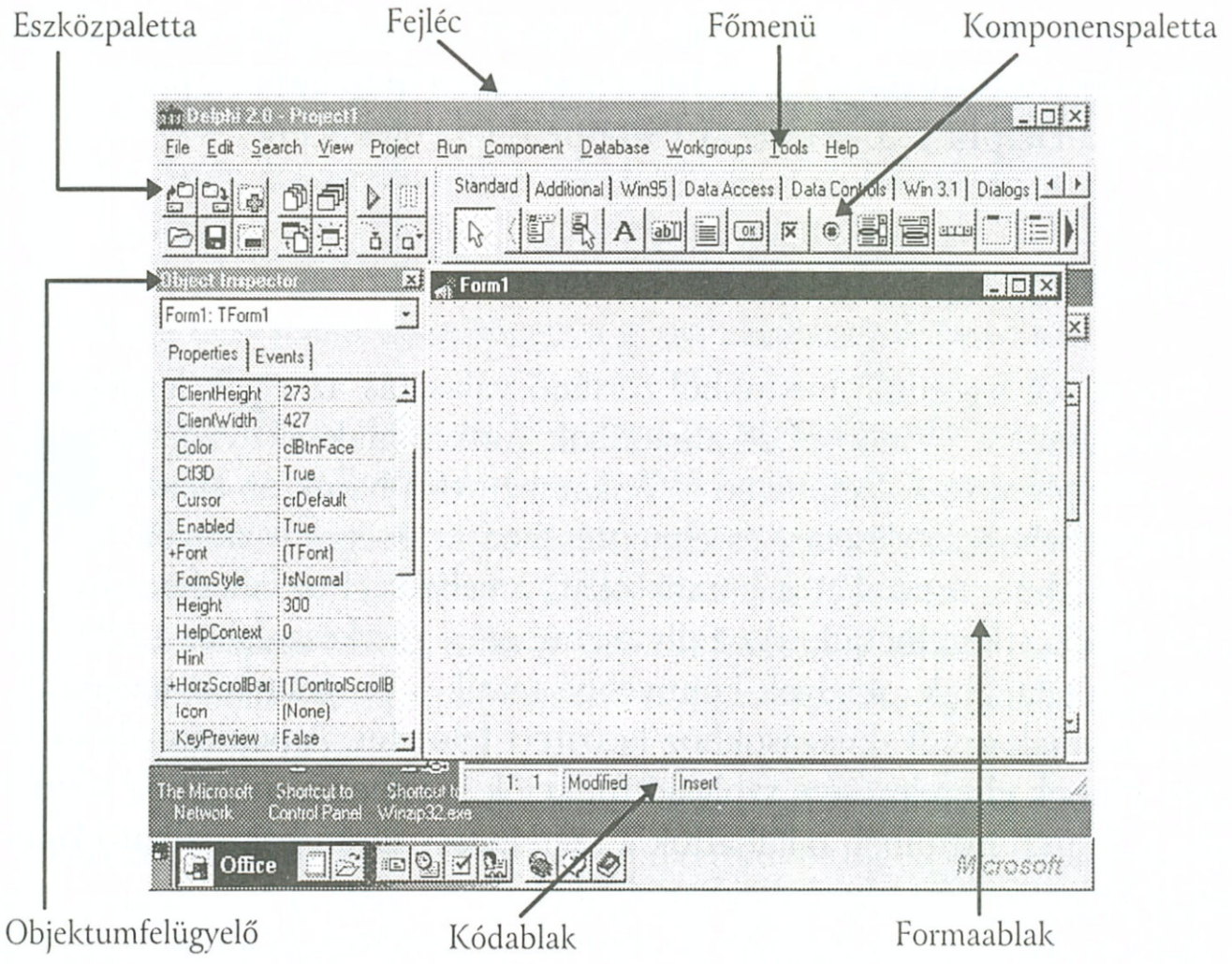
A Delphi eszköztárának gazdagsága könnyen kaotikussá teheti elképzeléseinket. A zűrzavar csökkentése érdekében, a fejezetben részletes leírás található a környezet nyújtotta lehetőségekről.

■ *A fő képernyőfelület áttekintése*

A Delphi elindításakor a képernyőn megjelenik a program verziószáma. Néhány másodperc múlva a 2-1. ábrán látható Delphi környezetben találjuk magunkat. A standard fejlesztői környezet jól elkülöníthető négy nagy részből áll: a menürendszerből, az objektumfelügyelőből, a form- és a (részben eltakart) kódszerkesztőből. Természetesen más, speciális célokat szolgáló ablakok is elérhetők az ezeket igénylő munkák (pl. hibakeresés) alkalmával.

Megjegyzés: *A Delphi megjegyzi az előző alkalommal beállított képernyőelrendezést, és azt megtartva jelentkezik be. Ezért elképzelhető, hogy a 2-1. ábrától különböző elrendezést kapunk.*

Most következzen a főablak leggyakrabban használt részeinek leírása. Az azt követő fejezetrészek a súgórendszert és a legtöbbet választott menüpontokat mutatják majd be.



2-1. ábra. A Delphi első bejelentkezésekor látható képernyő

A fejléc

Minden Microsoft Windows alkalmazás rendelkezik címsorral vagy fejléccel, amely az ablak legtetején helyezkedik el, és a program nevét tartalmazza. A felhasználó és a címsor közti kölcsönhatásokat nem az alkalmazás, hanem a Windows kezeli. Az alkalmazások ablakának a főmenü (vagy ha nincs, akkor a fejléc) alatti részét *kliensterületnek* nevezzük. A kliensterületre helyezett objektumok megjelenési módjáért és viselkedéséért egyedül az alkalmazás felelős.

Indításkor a Delphi címsorában a **Delphi 2.0 - Project1** felirat látható. Ez is jellemző a Microsoft Windows alkalmazásokra. A bonyolultabb programoknál (mint pl. a Delphi), amelyeknek több állapotuk lehet, a felirat a különböző állapotoknak megfelelően változik. Például, ha lefuttatunk egy

projektet a címfelirat **Delphi 2.0 - Project1 [Running]**-ra változik. Ha pedig hibajavítás (debugging) közben ideiglenesen megállítjuk a projektet, a fejlécben a **Delphi 2.0 - Project1 [Stopped]** lesz olvasható.

A főmenü

A főmenüből legördülő menükből történő választás az egyik legelterjedtebb módszer a Windows alkalmazások funkcióinak aktivizálására. Ez igaz a Delphire is. A Delphiben a főmenüben találhatók az alkalmazások fejlesztéséhez, teszteléséhez és elmentéséhez szükséges eszközök. A File menü tartalmazza azokat a parancsokat, amelyekkel az alkalmazást felépítő fájlokat kezelhetjük. Az Edit menüben a szerkesztéshez szükséges eszközöket találjuk, amelyek könnyebbé teszik a programkód megírását, és aktivizálják az alkalmazásokhoz készített felületet. A pozicionáló és az egyes elemek elrendezésére szolgáló funkciók is ehhez a menühöz tartoznak. A Search menüben találhatjuk a szerkesztés során használható keresést és helyettesítést végző dialógusablakokat.

A View menü gyors hozzáférést biztosít a projektek különböző részeihez és a Delphi környezet ablakaihoz. A Projekt menü a fordítással és a szintaxis-ellenőrzéssel kapcsolatos eszközöket tartalmazza. A Run menü lehetővé teszi az alkalmazás tesztelését fejlesztés alatt, és hozzáférést nyújt a hibajavításnál használatos eszközökhöz. A 9. fejezet részletesen tárgyalja a hibajavítási technikákat. A Component menü segítségével a Delphi komponens könyvtárához férhetünk hozzá. Többek között itt történik az új (vett vagy készített) komponensek telepítése is. A Database menü olyan eszközöket tartalmaz, mint a Database Form Expert (adatbázis-kezelő form készítő), amely megkönnyíti az adatbázis-kezelők készítését. A Tools menü tartalmazza a fontos Options menüpontot. Az Options pont választásával nyithatjuk meg azt a fülekkel ellátott dialógusablakot, amelyben a Delphi környezetének beállításain változtathatunk. A Tools menüben található még sok más hasznos eszköz, köztük a Database Desktop is. Nem maradhat ki a Help menü sem, amelyen keresztül a Delphi részletes súgórendszerébe léphetünk be.

Mint minden Windows alkalmazásban, itt is hozzáférhetünk az adott menüponthoz az ALT és az aláhúzott betű billentyűjének együttes lenyomásával (ezt hívják gyorsítóbillentyűnek). Ha csak önmagában az

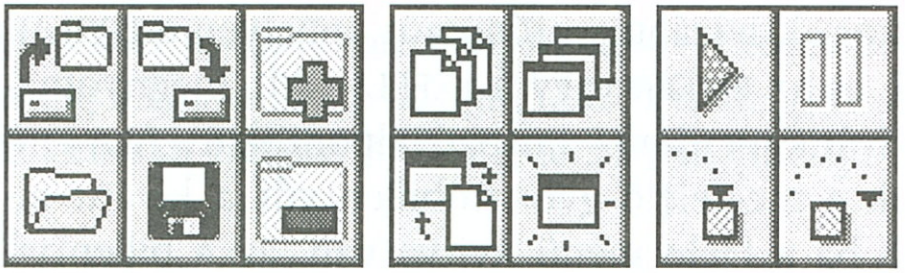
ALT gombot nyomjuk le, akkor aktivizáljuk a főmenüt, amelyen a NYILAK segítségével tudunk mozogni, és az ENTER vagy a LEFELE-NYÍL megnyomásával jeleníthetjük meg a kívánt menühöz tartozó alpontokat. A menü kinyitása után használhatjuk a megfelelő gyorsítóbillentyűt a kívánt alpont kiválasztására. Például, ha a Help menü nyitva van, az A billentyű lenyomásával megjelenik az About (névjegy) ablak. A kis- és nagybetűk használata nem jelent különbséget a gyorsítóbillentyűk esetében.

Tipp: Ha az About ablak nyitva van, megkaphatjuk a verziószámot, ha az ALT billentyűt lenyomjuk, és közben begépeljük a ***version*** *szót. Hasonlóképpen, ha a* ***team*** *szót írjuk be, megjelennek a Borland cég azon tagjainak nevei, akiknek a Delphi köszönhető. Ha a* ***team*** *helyett* ***developers****-t írunk, akkor a Borland fejlesztőgárdájának listáját láthatjuk. Az ALT* ***and*** *kombináció Anders Hejlsberg a Delphi atyjának képét hozza elő.*

Néhány funkció közvetlenül egy billentyűkombináció (parancsbillentyű) segítségével is elérhető, anélkül, hogy meg kellene nyitnunk a funkciót tartalmazó menüt. Pédául az ALT F4 lenyomásával – mint minden valamirevaló Windows alkalmazásban – kiléphetünk a Delphiből anélkül, hogy a File menüt kellene használnunk.

Az eszközpaletta

Az eszközpaletta (l. ábra, 44. old.) közvetlenül a főmenü alatt helyezkedik el. A Borland cég is beépített a termékébe egy a Windows alkalmazásoknál gyakran használt elemet, az eszközpalettát (SpeedBar-t). Az eszközpaletta ikonjaira történő kattintás előhívja a megfelelő funkciókat a menü használata nélkül. Mivel minden eleméhez tartozik egy billentyűkombináció is, a felhasználó – ízlésére hagyatkozva – dönthet, hogy melyik módszert választja. Ha a kurzort egy ikon fölé állítjuk, akkor egy keretben lévő címke (mostantól súgócímke) jelenik meg, amely a nyomógomb funkciójának azonosítására szolgáló rövid szöveget tartalmaz, ennek segítségével könnyebben el tudunk igazodni a különböző ikonok között. A 2-1. táblázatban láthatók az egyes gombok és rendeltetésük leírása.

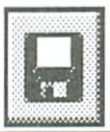
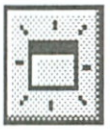


Megjegyzés: A 2-1. táblázatban az eredeti alapbeállított nyomógombok találhatók. Az eszközpalettát igényeinknek megfelelően alakíthatjuk, hozzáadhatunk és elvehetünk komponenseket. A hozzáadás módja a következő: helyezzük a kurzort az eszközpalettára, nyomjuk le a jobb oldali egérgombot, a megjelenő menüből válasszuk a Properties pontot majd a megjelenő dialógusablak instrukcióinak követésével helyezzük („fogd és vidd" módszerrel) a kívánt gombot a komponenspalettára.

2-1.táblázat. Az alapértelmezett eszközpaletta ikonjai

Ikon	*Név*	*Leírás*
	Open Project	Megnyit egy létező projektet (l. Programok futtatása és behívása című fejezetrész). Ugyanaz, mint a File menü Open pontja.
	Save All	Elmenti az adott Delphi projektet. Ugyanaz, mint a File menü Save All pontja. Ha a projekt új, akkor új fájlneveket kér be.
	Add File to Project	Lehetővé teszi egy korábban létrehozott form (testre szabható ablak) vagy Unit (forráskód) állomány projekthez csatolását. Megegyezik a File menü Add to Project pontjával.
	Select Unit from List	Aktivizálja a View Unit (Unit megjelenítő) dialógusablakot. Ugyanaz, mint a View menü Units pontja.
	Select Form from List	Aktivizálja a View Form (Form megjelenítő) dialógusablakot. Megegyezik a View menü Forms pontjával.
	Run	Lefuttatja az alkalmazást. Megegyezik a Run menü Run pontjának választásával (vagy az F9 lenyomásával).
	Pause	Megállítja a futó projektet. (A programokat általában a Run menü Run pontjának választásával vagy az F9 lenyomásával lehet folytatni.) Megegyezik a Run menü Pause pontjával.

2-1. táblázat. Az alapértelmezett eszközpaletta ikonjai (folytatás)

Ikon	*Név*	*Leírás*
	Open File	Megnyit egy létező form vagy Unit fájlt (bármilyen más fájl is megnyitható). Megegyezik a File menü Open File pontjával. Megjegyzés: ha egy form DFM (form tulajdonságok) állományát nyitjuk meg, akkor az a bináris formából automatikusan szöveggé konvertálódik, majd elmentéskor a Delphi visszaalakítja bináris fájllá.
	Save File	Elmenti a kiválasztott form vagy Unit fájlt. Megegyezik a File menü Save pontjával.
	Remove File from Project	Eltávolít egy kiválasztott form vagy Unit fájlt a projektből. Megegyezik a File menü Remove from Project pontjával.
	Toggle Form/Unit	A fókuszt a formról az ahhoz tartozó Unitra helyezi. Megegyezik a View menü Toggle Form/Unit pontjával (F12).
	New Form	Segítségével adhatunk új formokat (testre szabható ablakokat) a projekthez. Megegyezik a File menü New Form pontjával. Az 5. fejezet bővebb információkkal szolgál arról, hogyan használjunk több formot az alkalmazásban.
	Trace Into	Segítségével soronként futtathatjuk a projekt kódját. Ez egyben hibakereső eszközként (l. 9. fejezet) is szolgál. Megegyezik a Run menü Trace Into pontjával (F7).
	Step Over	Segítségével soronként futtathatjuk a projekt kódját, miközben az eljárásokat egy lépésben hajtja végre. Hasonlóan a Trace Into-hoz, ez is egy hibakereső eszköz (l. 9. fejezet). Megegyezik a Run menü Step Over pontjával (F8).

A komponenspaletta

A képernyő jobb felső részénél, közvetlenül az eszközpaletta jobb oldalán elhelyezkedő lapozható komponenspaletta (l. 2-1. ábra), amely körülbelül 100, az alkalmazásfejlesztést segítő alapeszközt tartalmaz. Az előző fejezetben ezek közül néhányat működés közben is láthattunk. A 4. fejezetben lesz szó a többi komponensről is.

A kiindulási formablak

A kiindulási formablak foglalja el a képernyő közepének nagy részét. Ahogy az előző fejezetben már tapasztaltuk, itt lehet megtervezni az ablakot, amelyet később a felhasználók látni fognak. A formokat leíró kód, amely az alkalmazás látványrészét tartalmazza, általában bináris formában tárolódik, ami más alkalmazások számára nem olvasható. A form fájlok mindig .DFM kiterjesztéssel szerepelnek. A formokról és formablakokról további részleteket a 4. fejezetben lehet megtudni.

Megjegyzés: *A Delphi lehetővé teszi, hogy a form fájljainkat (.DFM) ASCII szövegfájlként (l. 4. fejezet) mentsük el.*

Visual Basic Tipp: *Nem túl nehéz a Visual Basic formok ASCII reprezentációját Delphi formok ASCII reprezentációjává alakítani. Abban az esetben, ha sok ilyet kell átalakítanunk, kényelmesebb lehet az EarthTrek (617) 273-0308 által árult konverziós program használata.*

Az objektumfelügyelő ablaka

Az objektumfelügyelő az az eszköz, amelyben beállíthatjuk az alkalmazásunk kezdeti megjelenését definiáló tulajdonságokat. Azonkívül itt történik a külső események – úgymint a felhasználó beavatkozásai – fogadását megvalósító programkód váz kiválasztása. Az objektumfelügyelő két oldalból áll: a Properties (tulajdonságok) és az Events (események) oldalból. Ezeket a megfelelő fülekre történő kattintással érhetjük el. A Properties oldalt használjuk a form és az ahhoz tartozó komponensek fejlesztési idejű tulajdonságainak beállítására. Az Events oldalt pedig arra, hogy megírjuk a kész eseménykezelő eljáráshoz tartozó kódot. Az objektumfelügyelővel nagyobb részletességgel a 4., 5. és 6. fejezetek foglalkoznak.

Visual Basic Tipp: *Az objektumfelügyelő Properties oldala hasonlít a Visual Basic Properties ablakához.*

Amennyiben azt szeretnénk, hogy az objektumfelügyelő állandóan látható legyen, akkor a következőket kell csinálni:

1. Mozgassuk a kurzort az objektumfelügyelőre, és kattintsunk az egér jobb oldali gombjával. Ekkor megjelenik a gyorsmenü.
2. Válasszuk ki a gyorsmenü Stay on Top pontját.

A kódszerkesztő ablak

A kódszerkesztő kezdetben a formablak mögött és enyhén alatta található. A formablak alatt látható a kódszerkesztő alsó széle. A kódszerkesztőben írjuk meg a programkódot, amely megmondja a Delphinek, hogyan kezelje az eseményeket. Az eseménykezelő segítségével – miként azt az 1. fejezetben is említettük – teheti a programozó a Delphit érzékennyé a felhasználó beavatkozásaira. A form eseménykezelőinek tárolása a form vizuális elemeitől külön történik, ezeket az úgynevezett unit fájlokban tárolja a Delphi. A unit fájloknak mindig .PAS kiterjesztésük van, ezzel is jelezve, hogy az általuk tartalmazott kódot Object Pascalban, a Delphiben használt nyelven írták. A unit fájlok mindig ASCII formátumúak. (A unitokról többet a 6. fejezetben találhatunk.)

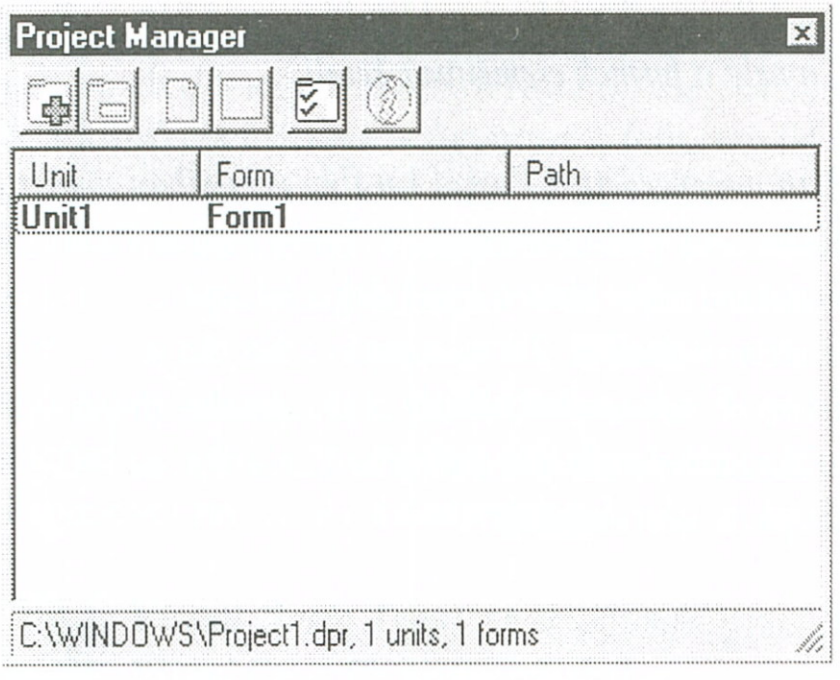
Visual Basic Tipp: *A unitok a Visual Basic moduljaihoz hasonlítanak. Egy nagy eltérés a Visual Basic-től az, hogy a Delphi formok nem tartalmazzák a kódot, amely a formok eseményeit kezeli.*

Pascal Tipp: *Általában megengedett a Pascal korábbi verzióiban megírt unitok használata (például, a Delphi támogatja a Turbo Pascal 7.0 WinCrt unitját), és majdnem teljes biztonsággal dolgozhatunk a Delphi 16-ban készített unitokkal is. Természetesen lehet, hogy változtatni kell a felületen, de minden kód, amely csak adatokat dolgoz fel, könnyen átvihető. Ellentétben a Delphi 16-tal a Turbo Pascal 7.0 OWL könyvtárát a Delphire nem vitték át.*

A Project Manager ablak

A Dephi alkalmazásokban lehetőség van (és ezt gyakran ki is használjuk) arra, hogy az alkalmazások közös kódrészleteket vagy előzőleg berendezett formokat használjanak, ezt könnyíti meg az, hogy a Delphi projektekbe szervezi az alkalmazásokat. Egy projekt a vizuális felületből és a felületet aktiváló kódból áll. A projektek több formot is tartalmazhatnak, ami többablakos alkalmazások készítését teszi lehetővé. A formok működéséhez szükséges kód (pl. a kód, amely eseményekre válaszol) egy külön unit fájlban tárolódik, amelyet a Delphi automatikusan összekapcsol a formmal. Az olyan általános programok kódjai is unit fájlokban vannak, amelyeket alkalmazásunk minden formja párhuzamosan használ(hat). Ilyen általános programrészletet célszerű külön tárolni, lehetővé téve bármikori újrafelhasználhatóságát, azaz a projekteinkhez való hozzászerkesztését.

Ez azt jelenti, hogy minden projekt egy vagy több form fájlból (.DFM kiterjesztésű fájlok) és egy vagy több unit fájlból (.PAS kiterjesztésű állományok) áll. A Project Manager ablakban nézhetjük meg a projekteket alkotó állományokat. A Project Managert a View menüben aktivizálhatjuk. Az ablakban a projektben használt form-unit párokat tartalmazó lista jelenik meg. Így néz ki a kiindulási Project Manager ablak:



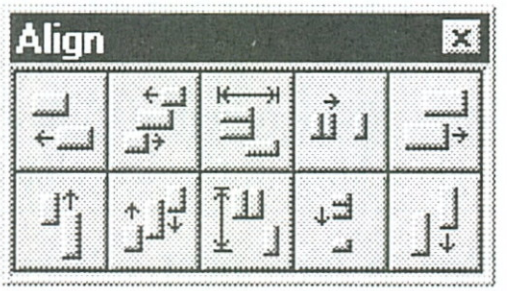
Az első sorban a kezdeti form és az ahhoz tartozó unit található.

A Delphi külön tárolja a projektet felépítő fájlokat, ezért célszerű nyomon követni az egyes állományok helyét. A Delphi létrehoz egy úgynevezett *projekt fájlt*, amely megmondja a Project Managernek (és nekünk is, ha megnézzük) a projektben szereplő fájlok elérési útvonalát. A projekt fájlt a File menü Save Project As pontjának (vagy az eszközpaletta ezzel egyenértékű Save Project gombjának) választásával hozhatjuk létre. A projekt fájloknak mindig .DPR kiterjesztésük van. A Save Project As választásakor a Delphi megnyit egy dialógusablakot, amelyben más néven (az előzőtől különböző) menthetjük el az aktuális projekt fájlunkat.

Tipp: *Ha az egyes projektek fájljait külön könyvtárba helyezzük, akkor megkönnyíthetjük a backup-ok készítését. (Az xcopy paranccsal az egész könyvtárat át lehet másolni.)*

Az Alignment Palette (elrendezéskezelő paletta)

Előfordulhat, hogy a form komponensekkel történő berendezése nehéz feladat elé állítja a programozót. Ha a View menü Alignment Palette pontját kiválasztjuk, akkor megjelenik az Align ablak. Ennek segítségével a komponensek egymáshoz és a formhoz viszonyított helyét határozhatjuk meg. A paletta a következőképpen néz ki:



Az elrendezéskezelő palettáról többet a 4. fejezetben tudhatunk meg.

A súgórendszer

A Delphit telepítő Setup program automatikusan felszereli a programot a szükséges segítségnyújtó állományokkal. A beépített súgórendszer jóval több információt tartalmaz, mint a Delphihez mellékelt kézikönyv. Ráadásul példaprogramokkal és tucatnyi hasznos táblázattal is elkényeztet bennünket. A különböző help állományokhoz más-más ikonokat hoz létre a Setup program.

A beépített súgórendszernek van egy rendkívül hasznos eleme: a helyzetérzékeny súgó. Ez azt jelenti, hogy bármikor, amikor szegítségre van

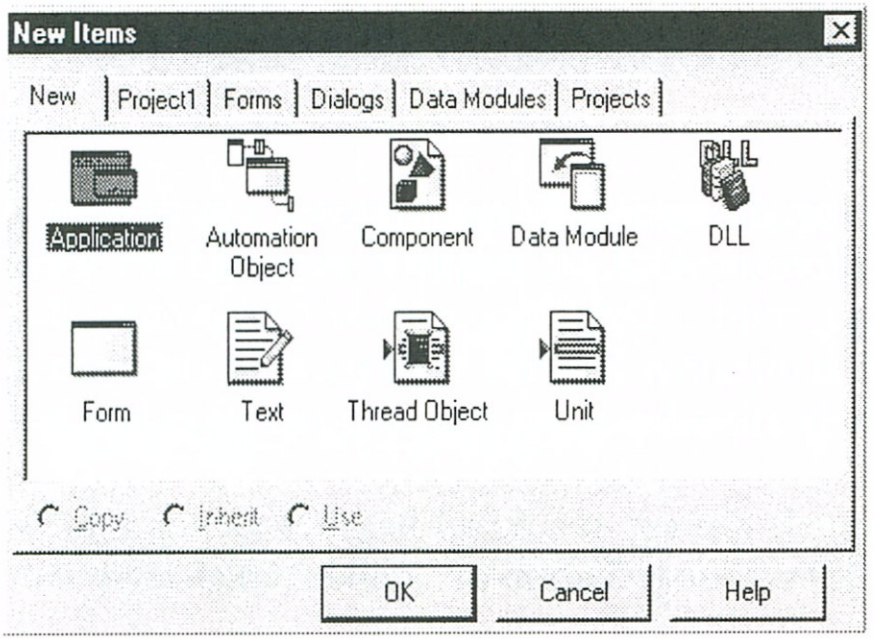
szükségünk, lenyomjuk az F1 billentyűt, így a Search dialógusablak kikerülésével juthatunk a kellő információk birtokába. A Delphi programnyelv minden kulcsszaváról, a hibakódokról és a Delphi környezet részeiről is felvilágosítást kaphatunk. A súgórendszer elindítása után a Help ablakot kívánság szerint mozgathatjuk, megváltoztathatjuk méretét, és elérhetjük, hogy csak egy ikon tanúskodjon ottlétéről.

■ *A File menüről részletesebben*

A File főmenüt akkor használjuk, ha a projekt fájljaival kívánunk dolgozni. Ez a menüpont felelős az állományok létrehozásáért, elmentéséért, behívásáért és nyomtatásáért. Ezek használatát röviden ismertetjük ebben a fejezetrészben. A Delphiből való kilépésnek több módja ismeretes, történhet ez is a File menün keresztül vagy az inputfókuszt a főablakra állítva az ALT-F4 billentyűkombináció lenyomásával. Egy másik Windows alkalmazásoknál használható módszer a rendszermenüre történő kettős kattintás vagy annak kinyitása után a Close pont választása.

A File menü legtöbb pontjának csak a fejlesztés megkezdése után vehetjük hasznát, ezek leírására a könyv későbbi fejezeteiben kerül sor. Most röviden bemutatjuk a File menü pontjait, ami remélhetőleg segíteni fogja a köztük történő eligazodást.

New. A New pont választásával megjelenik az alább látható New Items dialógusablak:



Ha az Application ikonra kattintunk, akkor az aktuális projektünk helyébe egy teljesen új (üres) projekt kerül. Ha az utolsó elmentés óta bármiféle változtatást végeztünk a projekten, akkor felbukkan egy dialógusablak, azt kérdezve, hogy el kívánjuk-e menteni munkánkat. Igen válasz esetén egy másik dialógusablak tűnik fel, amelyben a fájljainkat elmenthetjük. A többi ikon segítségével a projekthez új elemeket adhatunk, úgymint további formokat és unitokat (l. 5. fejezet). Az alábbi táblázat a dialógusablak oldalainak rövid leírását tartalmazza:

New	Ez az első oldal, amely az ikonokra történő kattintással létrehozható új egységekhez vizuális segítséget is nyújt.
Project1	Segítségével új formokat hozhatunk létre az előzőleg legyártott formjainkból objektumorientált technikák (öröklődés – l. 7. fejezet) használatával. Ez teszi lehetővé az aktuális formunk tulajdonságainak újbóli felhasználását is.
Forms	Ez az oldal tartalmazza az előregyártott formokat, amelyek közül válogathatunk, amikor a projektjeinkhez új formot kívánunk adni.
Dialogs	Hozzáférést nyújt az előregyártott dialógusablak formok széles skálájához.
Data Modules	Válogathatunk az adatbázisok fejlesztéséhez használt előregyártott adatkezelő formok közül.
Projects	Ez teszi lehetővé MDI-ket (multiple document interface=több dokumentumból álló felület) vagy SDI-ket (single document interface=egy dokumentumból álló felület) tartalmazó projektek készítését (l. 13. fejezet).

New Application. Ez egy másik lehetőség egy új alkalmazás elkezdésére, nyilvánvalóan egy kicsit gyorsabb, mint a New menüpont választása után a New Items dialógusablakban kiválasztani a New Application ikont.

New Form. Ez egy másik lehetőség új form hozzáadására a projekthez. (A több formot [multiform] használó projektekről bővebben az 5. fejezetben olvashatunk.)

New Data Module. Ez a funkció egy üres formot jelenít meg, amelyhez adathozzáférési komponenseket helyezhetünk el (l. 14. fejezet). Az adatmodul (Data Module) egy adatbázis-kapcsolatokat tartalmazó form, amely a projekt más adatkezelő formjaiban is használható.

Open. Hatására megjelenik egy dialógusablak, amelyben az aktuális projekthez már meglévő fájlokat adhatunk. Például ha egy létező unit fájlt (.PAS) nyitunk meg, a Delphi hozzáadja azt a projekthez.

Reopen. Hatására a leggyakrabban használt projektek listája jelenik meg.

Save. A Save pont elmenti azt a fájlt, amellyel éppen dolgoztunk. Az első alkalommal, amikor egy adott állományt el akarunk menteni, megjelenik egy dialógusablak, amelyben megadhatjuk a fájl nevét.

Save As. Ez a menüpont aktivizál egy dialógusablakot, amelyban lehetőségünk van a fájlt, amellyel dolgozunk egy másik név alatt elmenteni. A Save As így biztonsági másolatok készítésére is használható. Az egyetlen kényelmetlenséget az okozza, hogy ilyenkor a projekt nem a régi, hanem az új néven elmentett fájlt használja.

Save Project As. A Save Project As választás hatására megjelenik egy dialógusablak, amelyben új név alatt menthetjük el az aktuális projektet. Ezt egy új projekt fájl létrehozásával és a fájlok aktuális nevén történő elmentésével teszi. (Célszerű a Save As pontot használni, ha biztonsági másolat létrehozása céljából a projekt egyes részeinek új neveket szándékozunk adni.) A Save Project As pontot érdemes akkor is használni, ha a projektünkről backup másolatokat szeretnénk csinálni egy másik diszken, vagy a különböző verziókat el szeretnénk menteni. Ekkor megint gondot okozhat, hogy a Delphi az újonnan elmentett fájlokat fogja használni a régiek helyett.

Save All. Ez a választás az összes projekthez tartozó állományt elmenti, és létrehozza a kezdeti projekt fájlt, ha eddig nem létezett. (Emlékezzünk, hogy a projekt fájl tartalmazza a projektben használt fájlok listáját és egyéb, a Delphi által használt információkat.) Ha először használjuk ezt a pontot, akkor a Save Project As pontnál látott dialógusablak jelenik meg. (Először a unit neveket majd a projekt nevét kéri be.)

Close. Bezárja a fájlt, amelyikkel éppen dolgozunk.

Close All. Választása az aktuális projekt összes fájlját bezárja, miután felajánlotta a változások elmentését.

Use Unit. Megjelenít egy dialógusablakot, ami megkönnyíti a több unitot használó programok kezelését. (Emlékezzünk arra, hogy a Delphiben a unitok összefüggnek. Nekünk kell tájékoztatni a Delphit, hogy melyik unit használja egy másik unit funkcióit.)

Add To Project. Hatására megjelenik egy dialógusablak, amely segítségével egy formot vagy unit fájlt adhatunk az aktuális projekthez. (A különbség e pont és a File|Open között, hogy az utóbbi használatakor a fájl szerkeszthető, de nem lesz része a projektnek, így nem is lesz lefordítva.)

Remove From Project. Segítségével egy fájlt vehetünk ki a projektből.

Print. Kinyomtatja az aktuális formot vagy azt a unitot, amelyikkel éppen dolgozunk. A Print dialógusablak lehetőséget kínál bizonyos opciók beállítására és elfogadására, mielőtt az OK gombra kattintanánk.

Exit. Az Exit választása a Delphiből történő kilépés szokásos módja. Ha az aktuális projektet módosítottuk az utolsó elmentés óta, a Delphi megkérdezi, hogy elmentse-e a változásokat, mielőtt kilépnénk. Ekkor a mentésnél használt dialógusablakokat láthatjuk.

■ *Szövegszerkesztés*

A Delphit teljes képernyős editorral látták el. Mivel ez egy beépített – csupán programozási célokat szolgáló – szövegszerkesztő, hiányoznak az olyan funkciók, mint a sortörés és nyomtatásbeállítás, amelyekkel egy olyan primitív szövegszerkesztő is rendelkezik, mint a Microsoft Write. Viszont rendelkezik egy szintaxisellenőrző funkcióval, amely képes felismerni a programíráskor előforduló gyakoribb gépelési hibákat. A Delphi programszerkesztője színkóddal is ellátja a forráskód különböző részeit. Például a Delphi parancsok és a megjegyzések különböző színeket kapnak. A használt színeket az Environment Options dialógusablak (Tools | Options) Colors oldalán állíthatjuk be. A Delphi kódszerkesztő

szabványos Windows módszereket használ az olyan általános feladatokra, mint a beszúrás, felülírás, szövegkijelölés, másolás, kivágás, beillesztés és a törlés.

Tipp: *Az alapbeállításon kívül lehetőség van a szövegszerkesztő Brief, Epsilon és IDE Classic (ekvivalens a Turbo Pascal editorával) szabványoknak megfelelő testre szabására. (Lásd a beépített súgórendszer Editor Options oldalát.)*

Az Edit menü

Az Edit menü 17 pontot tartalmaz, amelyek aktivizálhatóságát az határozza meg, hogy a Delphi melyik részén van az inputfókusz. Alább található ezeknek egy rövid leírása.

Megjegyzés: *Az, hogy valamelyik elemen van az inputfókusz, azt jelenti a Windows-zsargonban, hogy a Windows figyelme arra az elemre összpontosul (az az elem fogadja a külső eseményeket). Általalában ez az az elem, amelyik reagál az egér kattintására, vagy billentyűlenyomásokra.*

Undo/Undelete, Redo. Az Undo visszaállítja az utolsó módosítás előtti helyzetet a kódszerkesztőben. Az Undelete visszaállítja az utolsó törlés előtti állapotot az adott formon. A Redo visszahozza az utolsó Undo előtti helyzetet. Az Undo-t és az Undelete-et a CTRL-Z kombinációval a Redo-t a SHIFT-CTRL-Z kombinációval is elérhetjük.

Cut, Copy, Paste. A Cut, Copy és Paste pontok használatára azután kerülhet sor, miután kijelöltünk egy szövegrészt vagy egy komponenst. A Cut (kivág) a kiválasztott szövegrészt a vágólapra (Clipboard) helyezi át, a Copy (másol) pedig egy másolatot készít a vágólapra, és a Paste bemásolja a vágólapon (clipboardon) lévő szöveget a Delphi alkalmazásba. Főleg akkor lehet ezeknek a funkcióknak hasznát venni, amikor információkat (szöveget vagy képet) cserélünk ki a Delphi és más Windows alkalmazás között. A Copy és Paste pontokat komponensekre vagy komponensek csoportjaira is alkalmazhatjuk.

Delete. A Delete eltávolítja a kijelölt részt anélkül, hogy másolatot készítene a vágólapra.

Select All. A Select All pont a kódszerkesztő ablak minden sorát, vagy egy form minden komponensét kijelöli.

Align to Grid. Segítségével az objektumok pontosan elhelyezhetőek a formokon. A 4. fejezet részletesebben foglalkozik a rács használatával.

Bring to Front, Send to Back. A Bring to Front pont a kiválasztott objektumot a többi fölé helyezi; a Send to Back pont a többi alá helyezi a kijelölt objektumot (l. 4. fejezet).

Align. Hatására megjelenik az Alignment (elrendezés) dialógusablak, amely segítségével a kijelölt komponenseket pontosan lehet pozicionálni egymáshoz és a formhoz képest. A View menü Alignment Palette (elrendezéskezelő paletta) pontja is a komponensek pozicionálását szolgálja. Az elrendezéskezelő paletta végig felül marad, amíg a változtatásokat végezzük a formon.

Size. A Size megnyit egy dialógusablakot, amelyben a kiválasztott komponensek szélességét és magasságát lehet beállítani (l. 4. fejezet).

Az Edit menü alábbi pontjai akkor elérhetőek, ha az inputfókusz valamelyik formon van.

Scale. Kinyit egy dialógusablakot, amelyben beállíthatjuk, hogy a kiválasztott formot és a rajta lévő komponenseket milyen mértékben kívánjuk nagyítani, ill. kicsinyíteni.

Tab Order. Menüpont kiválasztásának eredményeként megjelenik egy dialógusablak, amellyel egy formon vagy komponensen belüli látható komponensek esetében beállíthatjuk, hogy – a tabulátorbillentyű leütése esetén az inputfókusz milyen sorrendben haladjon végig rajtuk. (Bővebben lásd a 4. fejezetben.)

Creation Order. A Creation Order pont választásakor megjelenik egy dialógusablak, amellyel a nem vizuális komponensek esetében beállíthatjuk a létrehozás sorrendjét.

Lock Controls. Akkor érdemes ezt a menüpontot választani, ha elégedettek vagyunk egy adott formon lévő komponensek elhelyezésével, és nem akarjuk hogy ezen véletlenül változtassunk (a komponensek elmozgatásával).

Object. Ez a menüpont egy a formba beillesztett OLE objektum szerkesztéséhez és konvertálásához használható (l. 10. fejezet).

■ *A Search menü*

Ebben a szakaszban a Search menü rövid leírását követheti figyelemmel az olvasó. A Search menüt szövegrészek, hibák, objektumok stb. megtalálására lehet használni a kódszerkesztőben.

Find. Hatására kinyílik egy dialógusablak, amellyel a megadott szövegrészt kereshetjük meg.

Replace. Hatására megjelenik egy dialógusablak, amellyel kereshetjük és mással helyettesíthetjük a megadott szövegrészt.

Search Again. Megismétli az előző keresést.

Incremental Search. Ez egy elegáns módja a dialógusablak nélküli keresésnek. A begépelés közben megy a kurzor az addig beírt szöveg előfordulásának helyére.

Go to Line Number. Hatására megjelenik egy dialógusablak, amely a megadott sorszámú sorra helyezi a kurzort.

Find Error. Hatására a kurzor az utolsó fordításkor észlelt hiba helyére ugrik. Ez a menüpont csak sikertelen futtatás vagy fordítás után érhető el.

Browse Symbol. Megnyit egy dialógusablakot, amellyel a forráskódban egy megadott szimbólumot kereshetünk. Csak a már lefordított projekt esetében használható.

■ *A View menü*

A View menü olyan funkciókat tartalmaz, amelyek a környezet részeinek vagy a projektet alkotó komponensek és objektumok megjelenítésével, ill. elrejtésével kapcsolatosak. Alább található az egyes menüpontok rövid leírása. (A látható menüpontok listája függ attól, hogy éppen mit csinálunk a Delphiben.)

Project Manager. A Project Manager menüpont választásakor megjelenik a Project Manager ablak. A Projekt Manager-t a projektet alkotó fájljaink megtekintésére, törlésére, kiegészítésére vagy egy fájl jelenlegi projektünkbe történő elmentésére használhatjuk.

Project Source. A Project Source pont segítségével a projekt fájl forráskódját nézhetjük meg a kódszerkesztő ablakban.

Object Inspector. Az Object Inspector pont választásakor megjelenik az objektumfelügyelő ablak, lehetővé téve, hogy az egyes objektumok tulajdonságain és eseménykezelő eljárásain változtassunk. A 4. fejezet bővebben foglalkozik az objektumfelügyelő használatával.

Alignment Palette. Választása megnyitja az Align (elrendez) ablakot. Ezen az ablakon található eszköztárral gyorsan tudjuk a kijelölt alkotóelemeket rendezni egymáshoz, ill. a formhoz képest. A 4. fejezetben erről a pontról is többet tudhatunk meg.

Browser. A Browser menüpont választása megnyitja az Object Browser ablakot, amellyel megnézhetjük a projektben használt osztályok és metódusok hatáskörét, öröklődési hierarchiáját és hivatkozásait. Csak a már lefordított program esetében alkalmazható.

Breakpoints. A Breakpoints menüpont választásával megjelenik a Breakpoint List ablak. Információkat szerezhetünk az általunk elhelyezett töréspontokról. Az egér jobb oldali gombjának segítségével új töréspontokat hozhatunk létre, valamint átállíthatjuk a régieket. A 9. fejezetben további részleteket tudhatunk meg.

Call Stack. A Call Stack menüpont kiválasztásának hatására megjelenik a Call Stack ablak, amelyben az alkalmazásban lévő eljáráshívásokat és az eljárások paraméterátadásait követhetjük nyomon. Ez a menüpont hasznos eszköz a hibakeresésnél is (l. 9. fejezet).

Watches. A Watches menüpont kiválasztására megjelenő Watch List ablakban a programváltozókat követhetjük nyomon a program futása közben. Ez a menüpont is hasznos eszköz a hibakeresésnél (l. 9. fejezet).

Threads. Az éppen futó másodlagos folyamatot (thread), annak állapotát és elhelyezkedését jeleníti meg.

Component List. A Component List menüpont választásával komponenseket helyezhetünk el a kívánt formokon a billentyűzet használatával. Ezt vagy a Component List ablak Search mezejének – amelybe a kívánt komponens nevét kell begépelni – vagy a komponenseket tartalmazó lista segítségével tehetjük meg.

Window List. A Window list menüpont választásával megjelenik az összes Delphiben megnyitott ablak listája, amelyből kiválaszthatjuk, hogy melyik ablak legyen aktív. Ezt a funkciót az ALT-0 (nulla) kombinációval is előhívhatjuk.

Toggle Form/Unit. A Toggle Form/Unit menüpont kiválasztásával kapcsolgathatunk az aktuális form és a hozzá tartozó Unit között. (Az F12 lenyomása ugyanezt eredményezi.)

Units. A Units menüpont választásával megjelenik a View Units dialógusablak, amelyben a projektben szereplő összes unit van felsorolva. A kívánt unit nevére kettőt kattintva szerkeszthetjük azt a kódszerkesztőben. A menüpontot a CTRL-F12 lenyomásával is elérhetjük.

Forms. A Forms menüpont kiválasztásával a View Forms dialógusablak jelenik meg, amelyben a projektben szereplő összes form van felsorolva. Egy adott ablakmodell nevére kettőt kattintva az lesz az éppen tervezhető form. A menüpontot a SHIFT-F12 lenyomásával is elérhetjük.

New Edit Window. A New Edit Window választásakor egy új Code Editor (kódszerkesztő) ablak nyílik, amelybe az előző ablakban lévő kód töltődik. Ez praktikus módszer egy unit két különböző szakaszának párhuzamos szerkesztésére, hiszen a két szakaszt egy kódszerkesztőben nem lehetne egyszerre megjeleníteni.

Speedbar. A Speedbar ponttal tehetjük láthatatlanná, ill. láthatóvá az eszközpalettát. (Váltókapcsolóként működik.)

Component Palette. A Component Palette menüpont kiválasztásával tehetjük láthatóvá, ill. láthatatlanná a komponenspalettát. (Váltókapcsolóként működik.)

■ *A Project menü*

A Project menü nyolc menüpontot tartalmaz, ezek közül néhány a File menüben is elérhető (pl. az Add to Project és a Remove from Project pont). A többi pont leírása az alábbiakban történik.

Add to Repository. A Delphi 32 lehetővé teszi, hogy az általunk készített formokat hozzáadjuk a kelléktárához. A felvett formokat a New Items dialógusablak különböző oldalain láthatjuk viszont. A műveletet a menüpont választásakor megjelenő dialógusablak segítségével végezhetjük. (További információkhoz juthatunk, ha a dialógusablak Help gombjára kattintunk.)

Compile. A Compile menüpont segítségével bármely módosított kódot lefordíthatunk a projekt futtatása nélkül.

Build All. A menüpont választásával újrafordíthatjuk a projekt összes unitját. (A Compile-lal ellentétben ez nem csak a megváltozott kódot fodítja le.)

Syntax Check. Ellenőrzi a programkód szintaktikáját. Ez sokkal gyorsabb, mint lefuttatni és várni a nagyobb projektek esetén elkerülhetetlen hibaüzenetekre.

Information. A menüpont választásakor kinyílik egy dialógusablak, amely alkalmanként hasznos információkkal szolgál a lefordított kódról úgy, mint az .EXE fájl teljes mérete és arról, hogy a program különböző részei mekkora memóriát használtak el.

Options. A menüpont választásával megjelenik egy lapozható dialógusablak, amelyben a fordító és az általa lefordított kód beállításait adhatjuk meg. Például a dialógusablak Application oldalán beállíthatjuk a projekt nevét, ikonját és help-fájlja (.HLP) nevét. Az elkövetkezendő fejezetekből megtudhatjuk, hogyan lehet használni a dialógusablak oldalait.

■ *A Run menü*

Ezzel a menüponttal azt lehet beállítani, hogy a Delphi hogyan futtassa az aktuális projectet az integrált fejlesztői környezeten (IDE=integrated development environment) belül. Az első pont (Run) természetesen elég, ha csak le akarjuk futtatni a programunkat, de a többi menüpont is igen hasznos lehet. Ezeket általában hibakeresésre használják, és így a 9. fejezetben lesz részletesen szó róluk.

■ *A Component menü*

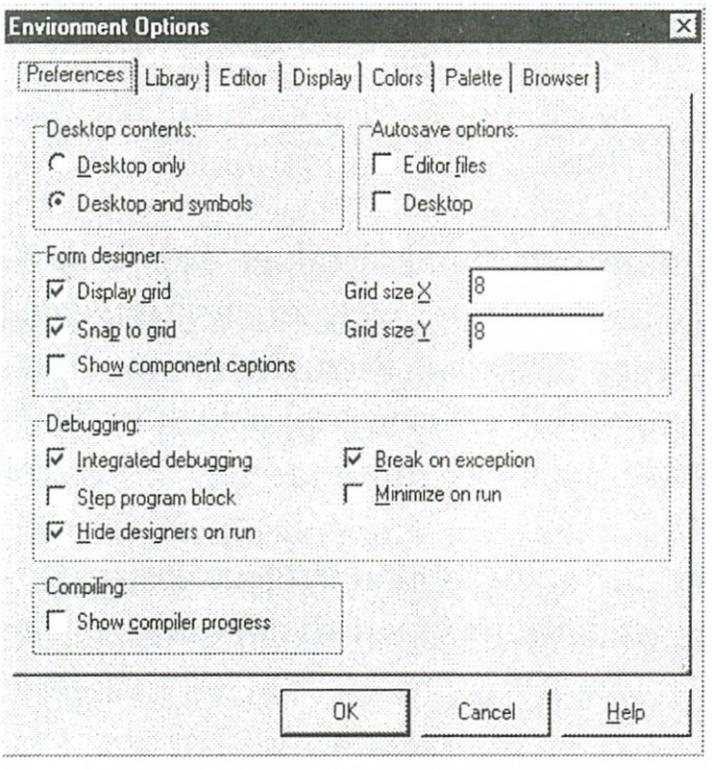
A Delphi egyik erős oldala a bővíthetőség. Ez a menü a komponenskönyvtárat kezeli. Például a New menüpont választásával egy dialógusablak jelenik meg, amely lehetővé teszi új komponensek készítését. (A részletekről l. a Delphihez mellékelt dokumentációt.) Az Install pont választásával kinyílik egy dialógusablak, amelynek segítségével egy már megírt komponenst telepíthetünk a komponenskönyvtárba. Az Open library ponttal egy új komponenskönyvtárat hozhatunk létre, így lehetőség nyílik arra, hogy több komponenspaletta közül választhassunk. A Rebuild library segítségével a komponenseken végzett változtatásokat állandósíthatjuk. Legvégül, a Configure Palette pont segítségével a Delphi komponenspalettáját szabhatjuk testre igényeink szerint.

■ *A Tools menü*

Ebben a fontos menüben a Delphi környezetének működésére vonatkozó beállításokat tehetjük meg. Például az Options menüpont megnyitja a lapozható Environment Options dialógusablakot, amelynek oldalain keresztül a Delphi részfunkcióit tudjuk befolyásolni. Például, az Editor oldal a Delphi kódszerkesztőjének működési módjáért felel, a Colors oldalon állíthatjuk be, hogy a Delphi a projekt különböző részeit milyen színekkel lássa el. Ez a létfontosságú dialógusablak a 2-2. ábrán látható. A következő fejezetekben gyakran feltűnik majd.

Tipp: *Mindenkinek ajánlom a dialógusablak Preferencies oldal Autosave options ablakrész alatt található Editor files alpont kijelölését, aminek eredményeképpen fájljaink automatikusan elmentődnek a projektek fordításakor. Ennek beállítása nélkül alkalmazást fejleszteni olyan, mint gumiszőnyeg nélkül tornázni: néha nagyon fájdalmas.*

Leggyakrabban a menü első két pontját használjuk. A második pont egy másik utat biztosít a (korábban megismert) kelléktárhoz való hozzáféréshez.



2-2. ábra. Az Environment Options dialógusablak

Most egy egyszerű példát láthatunk arra, hogyan lehet egy beállítást módosítani ebben a dialógusablakban. Változtassuk meg a tabulátor távolságok (tab stops) alapbeállítását 9 és 17 szóközről 4 és 8 szóközre! Ennek érdekében kövesse az alábbi pontokat:

1. Kattintsunk az Editor fülre.
2. A Tab Stops mezőbe írjunk 4-et, majd a SZÓKÖZ lenyomása után írjunk be 8-at.
3. Nyomjuk le az ENTERT.

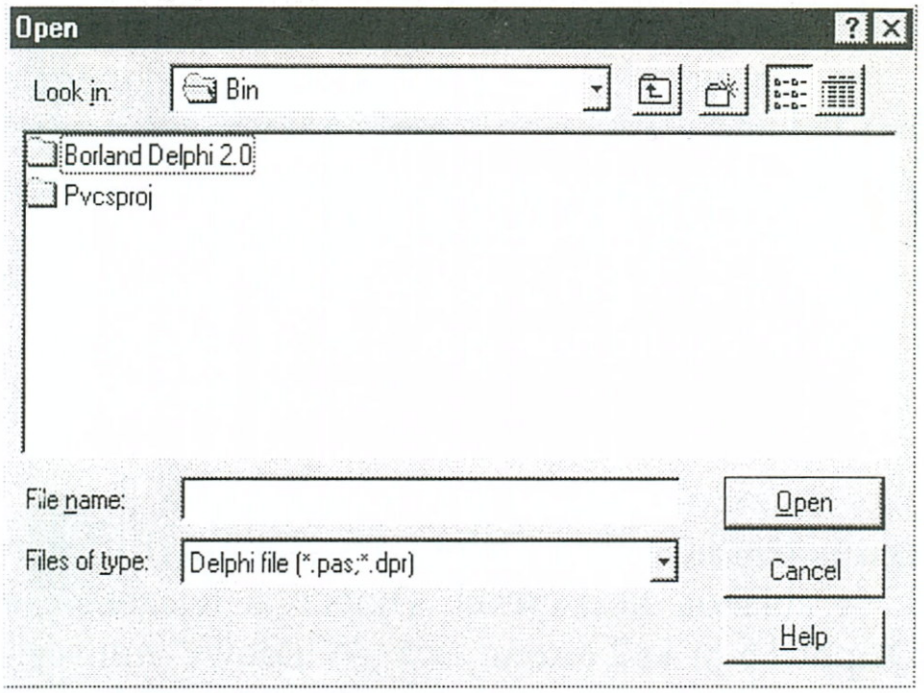
A dialógusablak többi pontja egy választási lehetőségeket felsoroló listát tartalmaz. Például az Editor Speedsetting listaablaka az Editor oldalon lehetőséget nyújt annak kiválasztására, hogy melyik billentyűkombináció milyen szerkesztési funkciónak feleljen meg. Itt csak négy lehetőség közül választhatunk: default (alapbeállítás), Classic (klasszikus: Turbo-Pascal-WordStar), Brief vagy Epsilon.

*A **Tools menüpont.*** Ez a menüpont hozzáférést biztosít a Delphi adott verziójához mellékelt eszközökhöz, többek között a Database Desktophoz, amely a Delphi összes verziójához jár, és amellyel bonyolult adatbázis-kezelő alkalmazásokat tesztelhetünk anélkül, hogy a szerverre szükségünk lenne. Az Add gombbal az általunk készített vagy vásárolt eszközöket telepíthetjük az eddigiek közé. Mint minden jó Windows alkalmazásnak, ezeknek az eszközöknek is saját súgórendszerük van.

■ *Programok betöltése és futtatása*

A Delphit nagyon sok és érdekes mintaprogrammal látták el. Fejezzük be ezt a fejezetet annak a folyamatnak az ismertetésével, amely a meglévő Delphi projektek futtatásához szükséges. Ehhez a Richedit demót használjuk fel. Ez a projekt jól mutatja, hogy milyen egyszerű a Delphi 32 új Windows 95 komponenseinek segítségével egy mini-szövegszerkesztőt csinálni.

Ha a File menü Open pontját választjuk vagy az Open Project ikonra kattintunk az eszközpalettán, akkor az itt látható Open dialógusablak jelenik meg:



Vegyük észre, hogy az ablakban nem látunk egyetlen állománynevet sem. Ennek az az oka, hogy a Delphi a felépítő fájlokat egy külön .DPR kiterjesztésű (ez a „Projekt fájl") állományban jegyzi az egyes projekteket, az ablakban látható könyvtárban (\Program Files\Borland\Delphi 2.0\Bin) ilyen nem található. Természetesen lehetőségünk van megváltoztatni a kiterjesztéseket – amelyekkel rendelkező állományokat a dialógusablak megjelenít – ha a File name mezőben új állományfajtát adunk meg, vagy új meghajtót választunk a dialógusablak megfelelő részében.

Váltsunk arra a meghajtóra és könyvtárra, amelyben a Delphi van (alapértelmezésben ez a /Program Files\Borland\Delphi 2.0\Bin\ könyvtár), és azután lépjünk be a Demos alkönyvtárba. Ennek alkönyvtárai közül válasszuk ki a „Richedit" nevűt, kettőt klikkeljünk rá, hogy a könyvtár belsejébe kerüljünk, ekkor a File ablakban meg kell hogy jelenjen a RICHEDIT.DPR fájl, amely a Richedit projektet alkotó állományok nevét és elérési útját tartalmazza. Kettőt kattintva erre, a Delphi beolvassa a Richedit projektet, amely egy Delphiben készített szövegszerkesztő.

Egy kis késéssel (várhatóan egy dialógusablak felbukkanásával félbeszakítva, amely megkérdezi, hogy elmentse-e a jelenlegi munkánkat), a Delphi beolvassa a Richedit projektet. A projektet teljes mértékben Delphiben írták, így jól mutatja azokat a lehetőségeket, amelyek hamarosan az olvasó kisujjában lesznek. Az F9 lenyomásával elindíthatjuk a programot. (Kilépni, mint minden futó projektből, az ALT F4 billentyűkombinációval lehet.) Figyelem! Ez a projekt is csak néhány száz programsorból áll.

3. FEJEZET

A Delphi 16 programozói környezet

Ebben a fejezetben megismerkedhetünk a Delphi 16 (Delphi 1.0) környezetét alkotó menük és ablakok kezelésével. (Ebben a fejezetben a „Delphi" szón mindig Delphi 16-ot kell érteni, és a „Windows" minvégig Windows 3.X-et fog jelenteni.) Ha nem mozgunk teljesen otthonosan a Microsoft Windows alkalmazások világában, akkor ez a fejezet bizonyára segíteni fog. Végső soron a Delphi maga is egy jól megírt Windows alkalmazás, így annak minden építőköve a Windows programokra jellemzően viselkedik. Tapasztalt Windows felhasználók lehet, hogy a fejezet egyes részeinek kihagyása mellett döntenek. Fontos, hogy amíg nem vagyunk teljesen tisztában a Windows alkalmazásokkal, addig nem leszünk képesek a Delphi lehetőségeit teljes mértékben kihasználni. A fejezet végére rutint szerzünk a Delphi beépített súgójának, fájlkezelőjének és szerkesztési kellékeinek használatában. Ha személyes használatra készítünk alkalmazást a Delphiben, akkor nem szükséges megfelelnünk a Windows szabványoknak. Azonban, ha az alkalmazást mások is használják, a Windows szabványok követése megkönnyíti az alkalmazás kezelésének elsajátítását. Nekünk is ügyelnünk kell arra, hogy betartsunk olyan egyszerű dolgokat, amelyek már beleivódtak a Windows felhasználók vérébe, pl. hogy egyszeres kattintással kiválaszthatnak egy menüpontot, és két kattintással pedig aktivizálhatnak egy adott alpontot.

Megjegyzés: *A Windows 95 alkalmazások összegyűjtött irányelvei The Windows Interface Guidelines for Software Design (ISBN 1-55615-679-0) című Microsoft-kiadványban találhatók meg. Érdeklődjön a Microsoftnál vagy a könyvesboltokban az ár és az elérhetőség iránt.*

A Microsoft Windows irányelveknek megfelelő beállításokat a Delphi alapértelmezésben tartalmazza. Miként azt már az előző fejezetben is láthattuk, a komponensek a szokásos alakkal és építőelemekkel rendelkeznek, mozgathatóak és méretezhetőek, ahogy azt a felhasználó el is várja. Hasonlóképpen a menük is a megszokott módon reagálnak a külső behatásokra. Természetesen a Delphi nem erőlteti ránk ezeket az alapértelmezéseket, de tény, hogy nem túl jó ötlet azokon meggondolatlanul és ok nélkül változtatni.

Megjegyzés: *A Help menüben elérhető Interactive Tutors jó kiegészítőként szolgál az ebben a fejezetben leírtakhoz.*

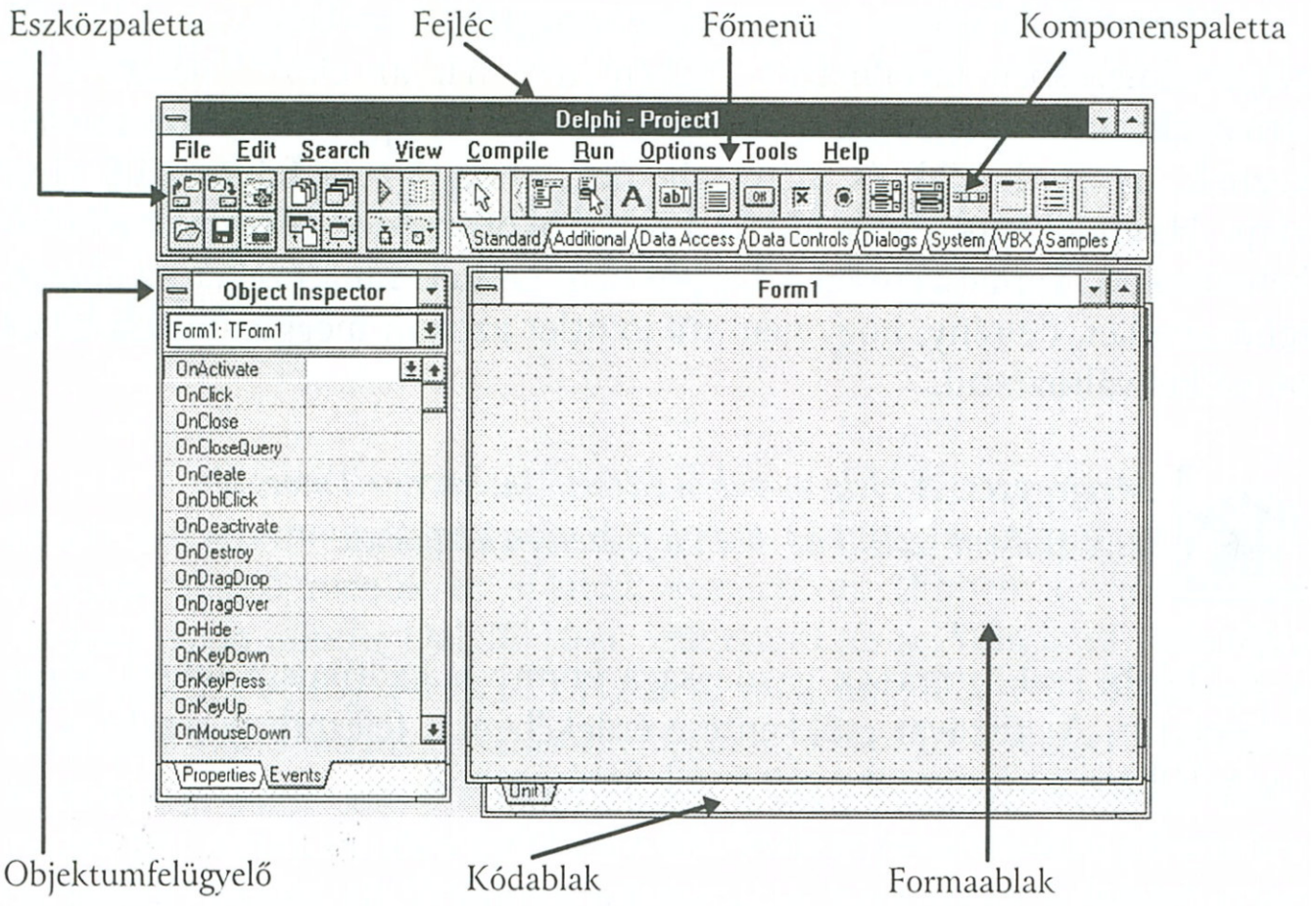
A Delphi eszköztárának gazdagsága könnyen kaotikussá teheti elképzeléseinket. A zűrzavar csökkentése érdekében, a fejezetben részletes leírás található a környezet nyújtotta lehetőségekről.

■ *A fő képernyőfelület áttekintése*

A Delphi elindításakor a képernyőn megjelenik a program verziószáma. Néhány másodperc múlva a 3-1. ábrán látható Delphi környezetben találjuk magunkat. A standard Delphi környezet négy, jól elkülöníthető nagy részből áll: a menürendszerből, az objektumfelügyelőből, a form- és a (részben eltakart) kódszerkesztőből. Más, speciális célokat szolgáló ablakok is elérhetők az ezeket igénylő munkák (pl. hibakeresés) alkalmával.

Megjegyzés: *A Delphi megjegyzi az előző alkalommal beállított képernyő-elrendezést és azt megtartva jelentkezik be. Ezért elképzelhető, hogy a 3-1.ábrától különböző elrendezést kapunk.*

Most pedig következzen a főablak leggyakrabban használt részeinek leírása. Az azt követő fejezetrészek pedig felölelik a súgórendszer és a legtöbbet választott menüpontok leírását.



3-1. ábra. A Delphi első bejelentkezésekor látható képernyő

A fejléc

Minden Microsoft Windows alkalmazás rendelkezik címsorral vagy fejléccel, amely az ablak legtetején helyezkedik el, és a program nevét tartalmazza. A felhasználó és a címsor közti kölcsönhatásokat nem az alkalmazás, hanem a Windows kezeli. A Windows alkalmazások ablakának a főmenü (vagy ha nics, akkor a fejléc) alatti részét *kliensterületnek* nevezzük. Egyedül az alkalmazás felelős a kliensterületre helyezett objektumok megjelenési módjáért és viselkedéséért.

Indításkor a Delphi címsorában a **Delphi - Project1** felirat látható. Ez is jellemző a Microsoft Windows alkalmazásokra. A bonyolultabb programoknál (mint a Delphi), amelyeknek több állapotuk lehet, a felirat a különböző állapotoknak megfelelően változik. Például, ha lefuttatunk egy projektet a címfelirat **Delphi - Project1 [Running]**-ra változik. Ha pedig hibajavítás (debugging) közben ideiglenesen megállítjuk a projektet, a fejlécben a **Delphi - Project1 [Stopped]** lesz olvasható.

A főmenü

A főmenüből legördülő menükből történő választás az egyik legelterjedtebb módszer a Windows alkalmazások funkcióinak aktivizálására. Ez igaz a Delphire is. A Delphiben a főmenüben találhatók az alkalmazások fejlesztéséhez, teszteléséhez és elmentéséhez szükséges eszközök. A File menü tartalmazza azokat a parancsokat, amelyekkel az alkalmazást felépítő fájlokat kezelhetjük. Az Edit menüben a szerkesztéshez szükséges eszközöket találjuk, amelyek könnyebbé teszik a programkód megírását és aktivizálják az alkalmazásokhoz készített felületet. A pozicionáló és elrendezést segítő eszközök is ehhez a menühöz tartoznak. A Search menüben vannak a szerkesztés során használható kereső és helyettesítő valamint a kódpozicionáló funkciók.

A View menü gyors hozzáférést biztosít a projektek különböző részeihez és a Delphi környezet ablakaihoz. A Compile menü a fordítással és a szintaxis ellenőrzéssel kapcsolatos eszközöket tartalmazza. A Run menü lehetővé teszi az alkalmazás tesztelését fejlesztés alatt, és hozzáférést nyújt a hibajavításnál használatos eszközökhöz. (A 9. fejezet részletesen tárgyalja a hibajavítási technikákat.) Az Options menü segítségével a Delphi környezetének működésére vonatkozó beállításokat tehetjük meg. A Tools menü külső eszközöket tartalmaz, úgymint ReportSmith, Database Desktop és az Image editor (képszerkesztő). Nem maradhat ki a Help menü sem, amellyel a Delphi részletes beépített súgórendszerébe léphetünk be.

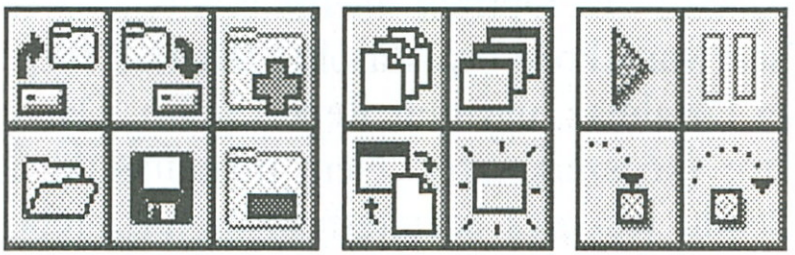
Mint minden Windows alkalmazásban, itt is hozzáférhetünk az adott menühöz az ALT és az aláhúzott betűnek megfelelő billentyűk együttes lenyomásával (ezt hívják gyorsítóbillentyűnek). Ha csak önmagában az ALT gombot nyomjuk le, akkor aktivizáljuk a főmenüt, amelyen aztán a NYILAK segítségével tudunk mozogni, az ENTER vagy a LEFELE-NYÍL megnyomásával megjeleníthetjük a kívánt menühöz tartozó alpontokat. A menü kinyitása után használhatjuk a megfelelő gyorsítóbillentyűt a kívánt alpont kiválasztására. Például, ha a Help menü nyitva van, az I billentyű lenyomása előhozza az Interactive Tutor-t (interaktív oktató). A kis- és nagybetűk használata nem jelent különbséget a gyorsítóbillentyűk esetében.

Néhány menüpont közvetlenül egy billentyűkombináció segítségével is elérhető anélkül, hogy meg kellene nyitnunk a menüpontot tartalmazó

menüt. Pédául az alt F4 lenyomásával – mint minden valamirevaló Windows alkalmazásban – kiléphetünk a Delphiből anélkül, hogy be kellene menni a File menübe (ha nincs más aktív ablak).

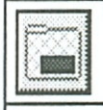
Az eszközpaletta

Az eszközpaletta (l. alábbi ábra) közvetlenül a főmenü alatt helyezkedik el. A Borland cég is beépített a termékébe egy a Windows alkalmazásoknál gyakran használt elemet, az eszközpalettát (SpeedBar-t). Az eszközpaletta ikonjaira történő kattintás előhívja a megfelelő funkciót a menü használata nélkül. Mivel az eszközpaletta minden eleméhez tartozik egy billentyűkombináció is, a felhasználó – ízlésére hagyatkozva – dönthet, hogy melyik módszert választja. Ha a kurzort egy ikon fölé állítjuk, akkor egy keretben lévő címke jelenik meg, amely a nyomógomb funkciójának azonosítására szolgáló rövid szöveget tartalmaz, ennek segítségével könnyebben el tudunk igazodni a különböző ikonok között. A 3-1. táblázatban láthatók az egyes gombok és rendeltetésük leírása.



Megjegyzés: A 3-1. táblázatban az eredeti alapértelmezett nyomogombok találhatók. Az eszközpalettán igényeinknek megfelelő változtatásokat végezhetünk. Például egy új gomb hozzáadásának módja a következő: helyezzük a kurzort az eszközpalettára, nyomjuk le a jobb oldali egérgombot, a megjelenő menüből válasszuk a Configure pontot majd a SpeedBar Editor dialógusablak instrukcióit követve helyezzük („fogd és vidd" módszerrel) a kívánt gombot a komponenspalettára.

3-1. táblázat. Az alapértelmezett eszközpaletta ikonjai

Ikon	*Név*	*Leírás*
	Open Project	Megnyit egy létező projektet (l. Programok futtatása és behívása című fejezetrész). Ugyanaz, mint a File menü Open pontja.
	Save Project	Elmenti az adott Delphi projektet. Ugyanaz, mint a File menü Save Project pontja. Ha a projektet még nem mentettük el, akkor fájlnevet kér be.
	Add File to Project	Lehetővé teszi egy korábban létrehozott form (testre szabható ablak) vagy Unit (forráskód) állomány projekthez csatolását. Megegyezik a File menü Add File pontjával.
	Select Unit from List	Aktivizálja a View Unit (Unit megjelenítő) dialógusablakot. Ugyanaz, mint a View menü Units pontja.
	Select Form from List	Aktivizálja a View Form (Form megjelenítő) dialógusablakot. Megegyezik a View menü Forms pontjával.
	Run	Lefuttatja az alkalmazást. Megegyezik a Run menü Run pontjának választásával (vagy az F9 lenyomásával).
	Pause	Megállítja a futó projektet. (A programokat általában a Run menü Run pontjának válaztásával vagy az F9 lenyomásával lehet folytatni.) Megegyezik a Run menü Pause pontjával.
	Open File	Általában egy meglévő Unit vagy Form fájl megnyitására használjuk. Megegyezik a File menü Open File pontjával.
	Save File	Elmenti a kiválasztott form vagy Unit fájlt. Megegyezik a File menü Save File pontjával.
	Remove File from Project	Eltávolít egy kiválasztott form vagy Unit fájlt a projektből. Megegyezik a File menü Remove File pontjával.
	Toggle Form/Unit	A fókuszt a formról az ahhoz tartozó Unitra helyezi. Megegyezik a View menü Toggle Form/Unit pontjával (F12).
	New Form	Segítségével adhatunk új formokat (testre szabható ablakokat) a projekthez. Megegyezik a File menü New Form pontjával. Az 5. fejezet bővebb információkkal szolgál arról, hogyan használjunk több formot az alkalmazásban.
	Trace Into	Segítségével soronként futtathatjuk a projekt kódját. Ez egyben hibakereső eszközként (l. 9. fejezet) is szolgál. Megegyezik a Run menü Trace Into pontjával (F7).
	Step Over	Segítségével soronként futtathatjuk a projekt kódját, miközben az eljrásokat egy lépésben hajtja végre. Hasonlóan a Trace Into-hoz, ez is egy hibakereső eszköz (l. 9. fejezet). Megegyezik a Run menü Step Over pontjával (F8).

A komponenspaletta

A képernyő jobb felső részénél, közvetlenül az eszközpaletta jobb oldalán elhelyezkedő lapozható komponenspaletta (l. 3-1. ábra) több, mint 75, az alkalmazásfejlesztést segítő alapeszközt tartalmaz. Az előző fejezetben ezek közül néhányat működés közben is láthattunk. A 4. fejezetben lesz szó a többi komponensről is.

A kiindulási formablak

A kiindulási formablak foglalja el a képernyő közepének nagy részét Ahogy az előző fejezetben már láttuk, itt lehet megtervezni az ablakot, amelyet később a felhasználók látni fognak. A formokat leíró kód, amelyek az alkalmazás látványrészét tartalmazzák, általában bináris formában tárolódik, ami más alkalmazások számára nem olvasható. A form fájlok mindig .DFM kiterjesztéssel szerepelnek. A formokról és formablakokról további részleteket a 4. fejezetben lehet megtudni.

Megjegyzés: *A Delphi lehetővé teszi, hogy form fájljainkat (.DFM) ASCII szövegfájlként mentse el (l. 4. fejezet).*

Visual Basic Tipp: *Nem túl nehéz a Visual Basic formok ASCII reprezentációját Delphi formok ASCII reprezentációjává alakítani. Ha sok formot kell átalakítanunk, kényelmesebb lehet az EarthTrek (617) 273-0308 által készített konverziós program használata.*

Az objektumfelügyelő ablaka

Az objektumfelügyelő az az eszköz, amelyben beállíthatjuk az alkalmazásunk megjelenésének kezdeti tulajdonságait. Azonkívül itt történik a külső események – úgymint a felhasználó beavatkozásai – fogadását megvalósító programkód elkészítése is. Az objektumfelügyelő két panelből áll: a Properties (tulajdonságok) és az Events (események) oldalakból. Ezek között a megfelelő fülekre történő kattintással lapozhatunk.

A Properties oldalt használjuk a form és az ahhoz tartozó komponensek fejlesztési idejű tulajdonságainak beállítására. Az Events oldalt pedig arra használjuk, hogy megírjuk a kész eseménykezelő eljáráshoz tartozó kódot. Az objektum felügyelővel nagyobb részletességgel a 4., 5. és 6. fejezetek foglalkoznak.

Visual Basic Tipp: *Az objektumfelügyelő Properties oldala hasonló a Visual Basic Properties ablakához.*

Ha azt szeretnénk, hogy az objektumfelügyelő állandóan látható legyen, akkor a következőket kell csinálni:

1. Mozgassuk a kurzort az objektumfelügyelőre, és kattintsunk az egér jobb oldali gombjával, ekkor megjelenik a gyorsmenü.
2. Válasszuk ki a gyormenü Stay on Top pontját.

A kódszerkesztő ablak

A kódszerkesztő a formablak mögött és enyhén alatta található. A formablak alatt látható a kódszerkesztő alsó széle. A kódszerkesztőben írjuk meg a programkódot, amely megmondja a Delphinek, hogyan kezelje az eseményeket. Az eseménykezelő segítségével – miként azt az 1. fejezetben is említettük – teheti a programozó a Delphit érzékennyé a felhasználó beavatkozásaira. A form eseménykezelőinek tárolása a form vizuális elemeitől külön történik, ezeket az úgynevezett unit fájlokban tárolja a Delphi. A unit fájloknak mindig .PAS kiterjesztésük van, ezzel is jelezve, hogy a kódot, amit tartalmaznak az Object Pascalban, a Delphiben használt nyelven írták. A unit fájlok mindig ASCII formátumúak. (A unitokról többet a 6. fejezetben találhatunk.)

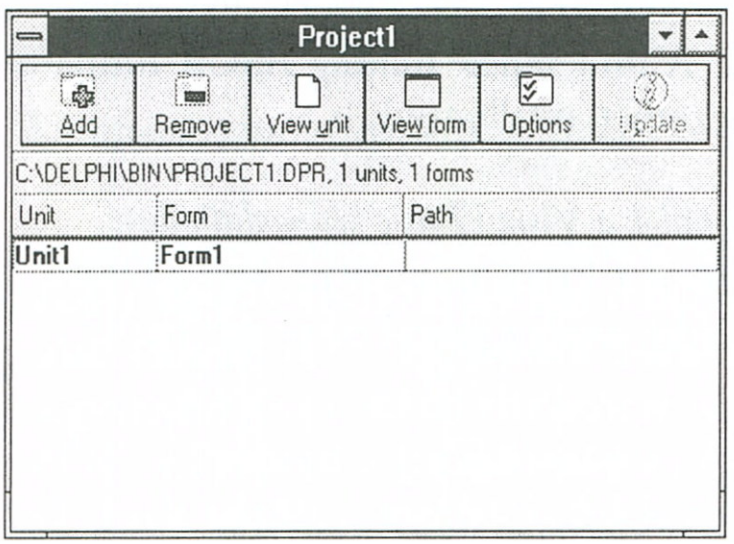
Visual Basic Tipp: *A unit fájlok a Visual Basic kódmoduljaihoz hasonlítanak. Nagy eltérést jelent a Visual Basic-től az, hogy a Delphi formok nem tartalmazzák a kódot, amely a formok eseményeit kezeli.*

Pascal Tipp: *Általában megengedett a Pascal korábbi verzióiban megírt unitok használata. (Például, a Delphi támogatja a Turbo Pascal 7.0 WinCrt unitját.) Természetesen lehet, hogy változtatni kell a felületen, de minden kód, amely csak adatokat dolgoz fel, könnyen átvihető.*

A Project Manager ablak

A Dephi alkalmazásokban lehetőség van (és ezt gyakran ki is használják) arra, hogy az alkalmazások közös kódrészleteket, vagy előzőleg berendezett formokat használjanak, ezt megkönnyíti, hogy a Delphi projektekbe szervezi az alkalmazásokat. Egy projekt a vizuális felületből és a felületet aktivizáló kódból áll. A projektek több formot is tartalmazhatnak, ami többablakos alkalmazások készítését teszi lehetővé. A formok működéséhez szükséges kód (pl. a kód, amely eseményekre válaszol) egy külön unit fájlban tárolódik, amelyet a Delphi automatikusan összekapcsol a formmal. Az olyan általános programok kódjai is unit fájlokban vannak, amelyeket alkalmazásunk minden formja párhuzamosan használ(hat). Ilyen általános programrészleteket célszerű külön tárolni, lehetővé téve bármikori újrafelhasználhatóságát, azaz a projekteinkhez való hozzászerkesztését.

Ez azt jelenti, hogy minden projekt egy vagy több form fájlból (.DFM kiterjesztésű fájlok) és egy vagy több unit fájlból (.PAS kiterjesztésű állományok) áll. A Project Manager ablakban nézhetjük meg a projekteket alkotó állományokat. A Project Managert a View menüben aktivizálhatjuk. Az ablakban a projektben használt form-unit párokat tartalmazó lista jelenik meg. Így néz ki a kiindulási Project Manager ablak:



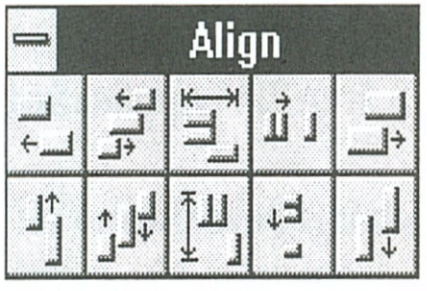
Az első sorban a kezdeti form és az ahhoz tartozó unit található.

A Delphi külön tárolja a projektet felépítő fájlokat, ezért célszerű nyomon követni az egyes állományok helyét. A Delphi létrehoz egy úgynevezett *projekt* fájlt, amely megmondja a Project Managernek (és nekünk is, ha megnézzük), hogy a projektben szereplő fájlokat hol lehet megtalálni. A projekt fájlt a File menü Save Project As pontjának (vagy az eszközpaletta ezzel egyenértékű Save Project gombjának) választásával hozhatjuk létre. A projekt fájloknak mindig .DPR kiterjesztésük van. A Save Project As választásakor a Delphi megnyit egy dialógusablakot, amelynek segítségével más néven (az előzőtől különböző) menthetjük el az éppen aktuális projekt fájlunkat.

Tipp: *Ha minden projekt fájljait külön könyvtárba helyezzük, akkor megkönnyíthetjük a backup-ok készítését. (Az xcopy paranccsal az egész könyvtárat át lehet másolni.)*

Az Alignment Palette (elrendezéskezelő paletta)

Előfordulhat, hogy a form-komponensekkel történő berendezése nehéz feladat elé állítja a programozót. Ha a View menü Alignment Palette pontját kiválasztjuk, akkor megjelenik az Align ablak. Ennek segítségével a komponensek egymáshoz és a formhoz viszonyított helyét határozhatjuk meg. A paletta a következőképpen néz ki:



Az elrendezéskezelő palettáról többet a következő fejezetben tudhatunk meg.

A súgórendszer

A Delphit telepítő Setup program automatikusan felszereli a programot 17 MB-ot kitevő segítségnyújtó állományokkal. A beépített súgórendszer jóval több információt tartalmaz, mint a Delphihez mellékelt kézikönyv. Ráadásul példaprogramokkal és tucatnyi hasznos táblázattal is elkényeztet bennünket. Minden súgóállományhoz külön ikonokat hoz létre a Setup program.

A beépített súgórendszernek van egy rendkívül hasznos eleme: a helyzetérzékeny súgó. Ez azt jelenti, ha bármikor segítségre van szükségünk, nyomjuk le az F1 billentyűt, aminek hatására a Search dialógusablak kikerülésével juthatunk a kellő információk birtokába. A Delphi programnyelv minden kulcsszaváról, a hibakódokról és a Delphi környezet részeiről is felvilágosítást kaphatunk. A súgórendszer elindítása után a Help ablakot kívánság szerint mozgathatjuk, megváltoztathatjuk méretét, és elérhetjük, hogy csak egy ikon tanúskodjon jelenlétéről.

Az alábbiak összefoglalják a Help menü alatt szereplő pontokat:

- **Contents.** A Delphi súgó rendszerének tartalomjegyzékét jeleníti meg.
- **Topic Search.** A menüpont választásakor megjelenik egy dialógusablak, amellyel a súgórendszerben kereshetünk segítséget egy megadott témában.
- **How to Use Help.** A menüpont magáról a help-ben való tájékozódásról ad felvilágosítást.
- **Interactive Tutors.** A menüpont egy párbeszédes oktatórendszert indít el. Nem sok mindent mutat be, de azért érdemes végigcsinálni.
- **Windows API.** A menüpont a Windows 3.1 API (Application Programming Interface) helpjét indítja el.
- **Database Form Expert.** A menüpont egy adatbázis-kezelő form létrehozását indítja el.
- **About.** A menüpont a Delphi névjegyét és verziószámát mutatja meg.

Tipp: Ha az About ablak nyitva van, megkaphatjuk a verziószámot, ha az ALT billentyűt lenyomjuk, és közben begépeljük a ***version*** *szót. Hasonlóképpen, ha ALT-ot és a* ***team*** *szót használjuk, akkor megjelennek a Borland cég embereinek nevei, akiknek a Delphi köszönhető.*

■ *A File menüről részletesebben*

A File menüt akkor használjuk, ha a projekt fájljaival kívánunk dolgozni. Ez a menüpont felelős a fájlok létrehozásáért, elmentéséért, behívásáért és nyomtatásáért. Ezek használatát röviden ismertetjük ebben a fejezetrészben. A Delphiből szintén a File menüben léphetünk ki.

(A kilépésnek több módja ismeretes, történhet ez is a File menün keresztül vagy az inputfókuszt a főablakra állítva az ALT-F4 billentyű kombináció lenyomásával. Egy másik Windows alkalmazásnál használható módszer a rendszermenüre történő kettős kattintás, vagy annak kinyitása után a Close pont választása.)

A File menü legtöbb pontjának csak a fejlesztés megkezdése után vehetjük hasznát, ezek leírására a könyv későbbi fejezeteiben kerül sor. Most röviden bemutatjuk ezeket a menüpontokat, ami remélhetőleg segíteni fogja a köztük történő eligazodást.

New Project. A New Project menüpont választásával új projektet hozhatunk létre. Az aktuális projektet a Delphi automatikusan törli a felületről. Ha ezen változtattunk az utolsó mentés óta, akkor megjelenik egy dialógusablak, és megkérdezi, hogy el kívánjuk-e menteni. Igen válasz esetén megjelenik egy másik dialógusablak, amelyben az elmentés történik.

Open Project. Az Open Project menüpont választásával egy korábban létrehozott projekten dolgozhatunk. Ha részletesebben szeretnénk látni, hogyan használjuk a menüpont által aktivizált dialógusablakot, akkor ne hagyjuk ki ennek a fejezetnek a végén található „Programok betöltése és futtatása" című részt.

Save Project. Ez a választás az összes projekthez tartozó állományt elmenti, és létrehozza a kezdeti projekt fájlt, ha eddig nem létezett. (Emlékezzünk, hogy a projekt fájl tartalmazza a projektben használt fájlok listáját és egyéb, a Delphi által használt információkat.) Ha először használjuk ezt a pontot, akkor a Save Project As pontnál látott dialógusablak jelenik meg. (Megjegyzés: először a unit neveket majd a projekt nevét kéri be.)

Save Project As. A Save Project As választás hatására megjelenik egy dialógusablak, amelyben új név alatt menthetjük el az aktuális projekthez tartozó összes fájlt. Ezt egy új projekt fájl létrehozásával és a fájlok aktuális nevén történő elmentésével teszi. A Save Project As pontot érdemes akkor is használni, ha a projektünkről backup másolatokat szeretnénk csinálni egy másik diszken, vagy a projekt különböző verzióit el szeretnénk menteni.

Close Project. Választása az aktuális projekt összes fájlját bezárja, miután felajánlotta a változások elmentését.

New Form. A New Form segítségével egy új formot illeszthetünk be a projektünkbe. (A több formot [multiform] használó projektekről bővebben az 5. fejezetben olvashatunk.)

New Unit. A New Unit segítségével olyan programkódot illeszthetünk be az alkalmazásunkba, amelyet elérhetővé kívánunk tenni a projekt minden részére. A Delphiben a kódot a unit fájlok tartalmazzák, amelyeket az egyes formokhoz köthetünk.

New Component. A New Component menüponttal megnyitjuk a Component Expertet, amely segítségével új komponenst hozhatunk létre. (További információkat a komponensek gyártásáról a Delphihez mellékelt dokumentációban találhatunk.)

Open File. Az Open File menüponttal egy már létező állományt nyithatunk meg szövegszerkesztőben. Ez a fájl nem kerül be a projektbe. A már létező formok és unit fájlok projektbe való beillesztésére az Add File menüpont szolgál.

Save File. A Save File pont elmenti az aktív formot vagy unitot. Az első alkalommal, amikor egy adott állományt el akarunk menteni, megjelenik egy dialógusablak, amelyben megadhatjuk a fájl nevét.

Save File As. A Save File As pont választásával megjelenik egy dialógusablak, amellyel az éppen szerkesztett formot vagy unitot menthetjük el új néven. Akkor is ezt a pontot érdemes használni, ha a projektünk egyes

részeiről biztonsági másolatokat szeretnénk csinálni, egy másik lemezre szeretnénk azokat elmenteni, vagy különböző verziókat kívánunk létrehozni. Érdemes még ezt a menüpontot választani, ha a jelenlegi alkalmazásunk egy részét későbbi projektekben kívánjuk használni. (Ebben az esetben a másik projektbe az itt elmentett állományt az Add File ponttal illeszthetjük be.)

Close File. A menüpont választásával az aktív formot vagy unitot zárhatjuk be. Kilépés előtt a változásokat elmenthetjük.

Add File. A menüpont választásakor megjelenik egy dialógusablak, amellyel egy már előzőleg elkészített munkánkat szerkeszthetjük az aktuális alkalmazáshoz. Ennek segítségével már létező formokat és unit fájlokat adhatunk a Delphihez.

Remove File. A menüpont segítségével az aktuális projektünkből kivehetünk egy kijelölt állományt. A fájl törlődik az alkalmazásból, viszont nem törlődik a diszkről, továbbra is a régi helyén marad. A lemezről való törléshez a Windows-ba beépített File Managert, vagy a megszokott DOS parancsokat használhatjuk.

Print. Kinyomtatja az aktuális formot vagy azt a unitot, amelyikkel éppen dolgozunk. A Print dialógusablak lehetőséget kínál bizonyos opciók beállítására, mielőtt az OK gombra kattintanánk.

Exit. Az Exit választása a Delphiből történő kilépésnek a szokásos módja. Ha az aktuális projektet módosítottuk az utolsó elmentés óta, a Delphi megkérdezi, hogy elmentse-e a változásokat mielőtt kilépnénk. Ekkor a mentésnél használt dialógusablakokat láthatjuk.

The History List. Ez a menüpont a három utoljára kinyitott Delphi projekt nevét tartalmazza. Ha az itt felsorolt fájlok közül egyre rákattintunk, a Delphi automatikusan betölti a projektet. Ez megkönnyíti a folyamatban lévő munkához való visszatérést.

■ *Szövegszerkesztés*

A Delphit teljes képernyős editorral látták el. Mivel ez egy beépített, csupán programozási célokat szolgáló szövegszerkesztő, hiányoznak az olyan funkciók, mint sortörés- és nyomtatásbeállítás, amelyekkel egy olyan primitív szövegszerkesztő is rendelkezik, mint a Microsoft Write. Másfelől viszont rendelkezik egy szintaxis ellenőrző funkcióval, amely képes felismerni a programíráskor előforduló gyakoribb gépelési hibákat. A Delphi programszerkesztője színkóddal is ellátja a forráskód különböző részeit. Például a Delphi parancsok és a megjegyzések különböző színeket kapnak. A használt színeket az Options menü Environment pontja alatt állíthatjuk be. A Delphi kódszerkesztő szabványos Windows módszereket használ az olyan általános feladatokra, mint a beszúrás, felülírás, szövegkijelölés, másolás, kivágás, beillesztés és a törlés.

Tipp: *Az alapbeállításon kívül lehetőség van a szövegszerkesztő Brief, Epsilon és IDE Classic (ekvivalens a Turbo Pascallal) szabványoknak megfelelő testre szabására. (L. a beépített súgórendszer Editor Options oldalát.)*

Az Edit menü

Az Edit menüpont 17 pontot tartalmaz, amelyek attól függően elérhetők, hogy a Delphi melyik részén van az inputfókusz. Alább található ezek rövid leírása.

Megjegyzés: *Az, hogy valamelyik elemen van az inputfókusz, azt jelenti a Windows-zsargonban, hogy a Windows figyelme arra az elemre összpontosul. Általalában ez az az elem, amelyik reagál az egér kattintására vagy billentyűlenyomásokra.*

Undo/Undelete, Redo. Az Undo visszaállítja az utolsó esemény előtti helyzetet a kódszerkesztőben. Az Undelete visszaállítja az utolsó törlés előtti állapotot az adott formon. A Redo visszahozza az utolsó Undo előt-

ti helyzetet. Az Undo-t és az Undelete-et a CTRL-Z kombinációval, a Redo-t a SHIFT-CTRL-Z kombinációval is elérhetjük.

Cut, Copy, Paste. A Cut, Copy és Paste pontok használatára azután kerülhet sor, miután kijelöltünk egy szövegrészt vagy egy komponenst. A Cut (kivág) a kiválasztott szövegrészt a vágólapra (clipboard) helyezi át, a Copy (másol) pedig egy másolatot készít a vágólapra, és a Paste bemásolja a vágólapon lévő szöveget a Delphi alkalmazásba. Főleg akkor lehet ezeknek a funkcióknak hasznát venni, amikor információkat (szöveget vagy képet) cserélünk ki a Delphi és más Windows alkalmazás között. A Copy és Paste pontokat komponensekre, vagy komponensek csoportjaira is alkalmazhatjuk (l. 4. fejezet).

Delete. A Delete eltávolítja a kijelölt részt anélkül, hogy másolatot készítene a vágólapra.

Select All. A Select All pont a kódszerkesztő ablak minden sorát, vagy egy form minden komponensét kijelöli.

Align to Grid. Segítségével az objektumok pontosan elhelyezhetőek a formokon. A 4. fejezet részletesebben foglalkozik a rács használatával.

Bring to Front, Send to Back. A Bring to Front pont a többi objektum fölé, a Send to Back pont a többi alá helyezi a kijelölt objektumot (l. 4. fejezet).

Align. Hatására megjelenik az Alignment (elrendezés) dialógusablak, amely segítségével a kijelölt komponenseket pontosan lehet pozicionálni egymáshoz és a formhoz képest. A View menü Alignment Palette (elrendezéskezelő paletta) pontja is ezt a célt szolgálja. Az elrendezéskezelő paletta végig felül marad (az összes ablak felett), amíg a változtatásokat végezzük a formon.

Size. A Size megnyit egy dialógusablakot, amelyben a kiválasztott komponensek szélességét és magasságát lehet beállítani (l. 4. fejezet).

Az Edit menü alábbi pontjai akkor elérhetőek, ha a fókusz valamelyik formon van.

Scale. Kinyit egy dialógusablakot, amelyben beállíthatjuk, hogy a kiválasztott formot és a rajta levő komponenseket milyen mértékben kívánjuk nagyítani ill. kicsinyíteni.

Tab Order. A menüpont kiválasztásának eredményeként megjelenik egy dialógusablak, amellyel egy formon vagy komponensen belüli látható komponensek esetében beállíthatjuk, hogy – a tabulátorbillentyű leütése esetén a fókusz milyen sorrendben haladjon végig rajtuk. (Bővebben lásd a 4. fejezetben.)

Creation Order. A Creation Order pont választásakor megjelenik egy dialógusablak, amellyel a nem látható komponensek esetében beállíthatjuk a létrehozás sorrendjét.

Lock Controls. Akkor érdemes ezt a menüpontot választani, ha elégedettek vagyunk egy adott formon lévő komponensek elhelyezésével, és nem akarjuk hogy ezen véletlenül változtassunk (mondjuk egy komponens elmozgatásával).

Object. Ez a menüpont a formba beszúrt OLE objektum szerkesztéséhez és konvertálásához használható.

■ *A Search menü*

Ebben a szakaszban a Search menü rövid leírását követheti figyelemmel az olvasó. A Search menüt szövegrészek, hibák, objektumok stb. megtalálására lehet használni a kódszerkesztőben.

Find. Hatására kinyílik egy dialógusablak, amellyel a megadott szövegrészt kereshetjük meg.

Replace. Hatására megjelenik egy dialógusablak, amellyel kereshetjük és mással helyettesíthetjük a megadott szövegrészt.

Search Again. Megismétli az előző keresést.

Incremental Search. Ez egy elegáns módja a dialógusablak nélküli keresésnek. A kurzor begépelés közben megy az addig beírt szöveg előfordulási helyére.

Go to Line Number. Hatására megjelenik egy dialógusablak, amely a megadott sorszámú sorra helyezi a kurzort.

Find Error. Hatására a kurzor az utolsó fordításkor észlelt hiba helyére ugrik. Ez a menüpont csak egy nem sikerült futtatás vagy fordítás után érhető el.

Browse Symbol. Megnyit egy dialógusablakot, amellyel a forráskódban egy megadott szimbólumot kereshetünk. Csak a már lefordított projekt esetében használható.

■ *A View menü*

A View menü olyan funkciókat tartalmaz, amelyek a környezet részeinek vagy a projektet alkotó komponensek és objektumok megjelenítésével ill. elrejtésével kapcsolatosak. Alább található az egyes menüpontok rövid leírása. (A látható menüpontok listája függ attól, hogy éppen mit csinálunk a Delphiben.)

Project Manager. A Project Manager menüpont választásakor megjelenik a Projekt Manager ablak. A Projekt Managert a projektet alkotó fájljaink megtekintésére, törlésére, kiegészítésére, vagy egy fájl jelenlegi projektünkbe történő elmentésére használhatjuk.

Project Source. A Project Source pont segítségével a projekt fájl forrás kódját nézhetjük meg a kódszerkesztő ablakban.

Object Inspector. Az Object Inspector pont választásakor megjelenik az objektumfelügyelő ablak, lehetővé téve, hogy az egyes objektumok tulajdonságain és az eseményekhez tartozó eljárásokon változtassunk. A 4. fejezet bővebben foglalkozik az objektumfelügyelő használatával.

Alignment Palette. Választása megnyitja az Align (elrendez) ablakot. Ezen az ablakon található eszköztárral gyorsan tudjuk a kijelölt alkotóelemeket rendezni egymáshoz ill. a formhoz képest. A 4. fejezetben erről a pontról is többet tudhatunk meg.

Browser. A Browser menüpont választása megnyitja az Object Browser ablakot, amellyel megnézhetjük a projektben használt osztályok és metódusok hatáskörét, öröklődési hierarchiáját és hivatkozásait. Csak a már lefordított program esetében alkalmazható (l. 7. fejezet).

Breakpoints. A Breakpoints menüpont választásával megjelenik a Breakpoint List ablak. Információkat szerezhetünk az általunk elhelyezett töréspontokról. Az egér jobb oldali gombjának segítségével új töréspontokat hozhatunk létre, valamint átállíthatjuk a régieket. A 9. fejezetben további részleteket tudhatunk meg.

Call Stack. A Call Stack menüpont kiválasztásának hatására megjelenik a Call Stack ablak, amelyben az alkalmazásban lévő eljáráshívásokat és az eljárások paraméterátadását követhetjük nyomon. Ez a menüpont hasznos eszköz a hibakeresésnél is (l. 9. fejezet).

Watches. A Watches menüpont kiválasztására megjelenő Watch List ablakban a programváltozókat követhetjük nyomon a program futása közben. Ez a menüpont is hasznos eszköz a hibakeresésnél (l. 9. fejezet).

Component List. A Component List menüpont választásával komponenseket helyezhetünk el a kívánt formokon a billentyűzet használatával. Ezt vagy a Component List ablak Search mezejének – amelyben a kívánt komponens nevét kell begépelni – vagy a komponenseket tartalmazó lista segítségével tehetjük meg.

Window List. A Window List menüpont választásával megjelenik az összes Delphiben megnyitott ablak listája, amelyből kiválaszthatjuk, hogy melyik ablak legyen aktív. Ezt a funkciót az ALT-0 (nulla) kombinációval is előhívhatjuk.

Toggle Form/Unit. A Toggle Form/Unit menüpont kiválasztásával kapcsolgathatunk az aktuális form és a hozzá tartozó unit között. (Az F12 lenyomása ugyanezt eredményezi.)

Units. A Units menüpont választásával megjelenik a View Units dialógusablak, amelyben a projektben szereplő összes unit van felsorolva. A kívánt unit nevére kettőt kattintva szerkeszthetjük azt a kódszerkesztőben. A menüpontot a CTRL-F12 lenyomásával is elérhetjük.

Forms. A Forms menüpont kiválasztásával a View Forms dialógusablak jelenik meg, amelyben a projektben szereplő összes form van felsorolva. Egy adott ablakmodell nevére kettőt kattintva az lesz az éppen tervezhető form. A menüpontot a SHIFT-F12 lenyomásával is elérhetjük.

New Edit Window. A New Edit Window választásakor egy új Code Editor (kódszerkesztő) ablak nyílik, amelybe az előző ablakban lévő kód töltődik. Ez praktikus módszer egy unit két különböző szakaszának párhuzamos szerkesztésére, hiszen a két szakaszt egy kódszerkesztőben nem lehetne egyszerre megjeleníteni.

Speedbar. A Speedbar ponttal tehetjük láthatatlanná ill. láthatóvá az eszközpalettát. (Váltókapcsolóként működik.)

Component Palette. A Component Palette menüpont kiválasztásával tehetjük láthatóvá ill. láthatatlanná a komponenspalettát. (Váltókapcsolóként működik.)

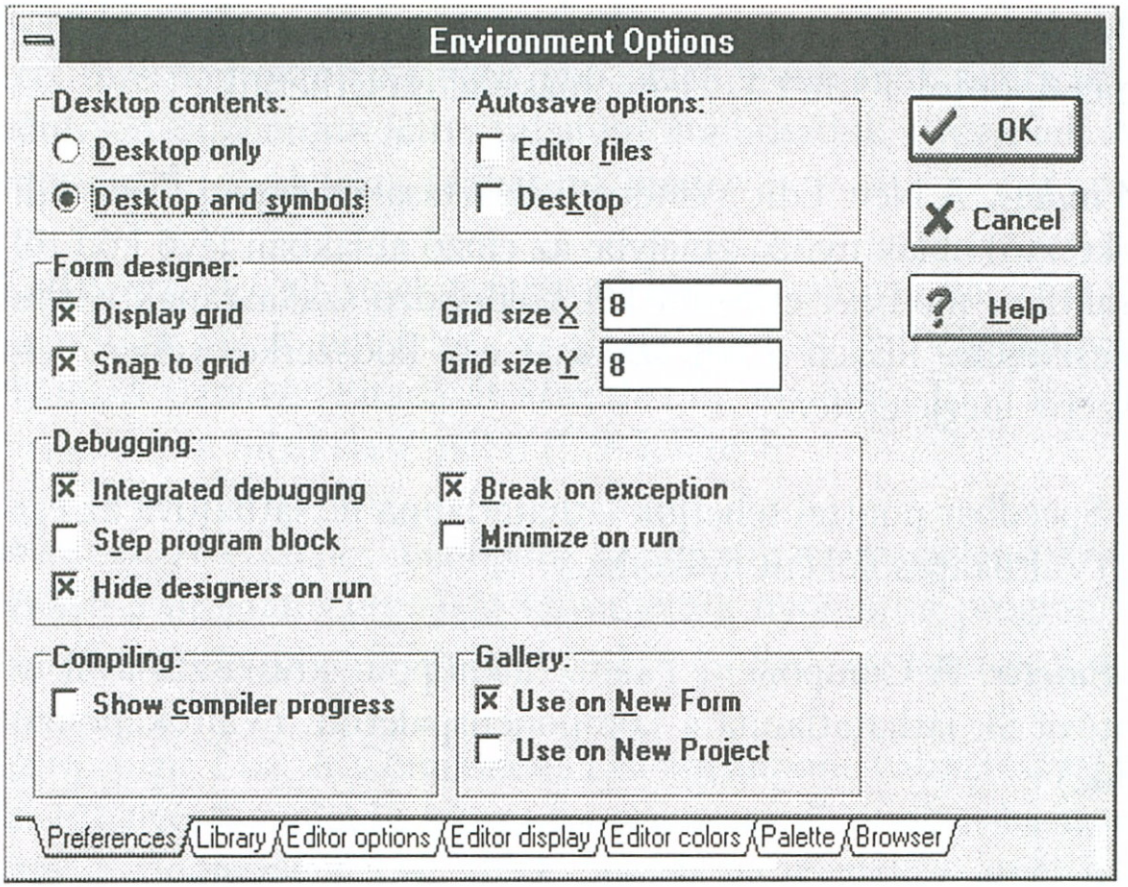
■ *Az Options menü*

Az Options menü 7 pontot tartalmaz. Ezek közül az első kettő a leggyakrabban használt. A harmadik, a Tools alpont ugyanaz, mint a Tools menüpont. Az utolsó három pontot akkor használjuk, ha könyvtárakkal dolgozunk (pl. amikor a Delphit komponensekkel kívánjuk kibővíteni).

Az első menüpont (Project) választásakor megjelenik egy lapozható ötoldalas dialógusablak. A dialógusablak legfontosabb részeivel majd a 6. fejezet foglalkozik. (A többi oldal megismeréséhez használjuk a beépített help-et.)

Az Environment menüpont választásakor a Delphi megnyitja a 3-2. ábrán látható dialógusablakot.

A különböző fülekre történő kattintással előjönnek az oldalak, amelyekben megváltoztathatjuk a felület színeit, bállíthatjuk a tabulátortávolságokat, vagy még általánosabban, a programozást kényelmesebbé tevő beállításokat tehetünk. Például megváltoztathatjuk a kódszerkesztő billentyűzetkiosztását. Azért, hogy lássunk egy példát valamelyik beállítás megváltoztatására, cseréljük ki az alapértelmezésben szereplő 9 és 17 szóköz hosszúságú tabulátortávolságot 4 és 8 szóköz hosszúságúra. Ezt a következő pontokkal valósíthatjuk meg:



3-2. ábra. Az Environment Options dialógusablak

1. Kattintsunk az Editor feliratú fülre.
2. A Tab Stops keretbe gépeljünk 4-et, majd a SZÓKÖZ lenyomása után 8-at.
3. Üssük le az ENTER-t.

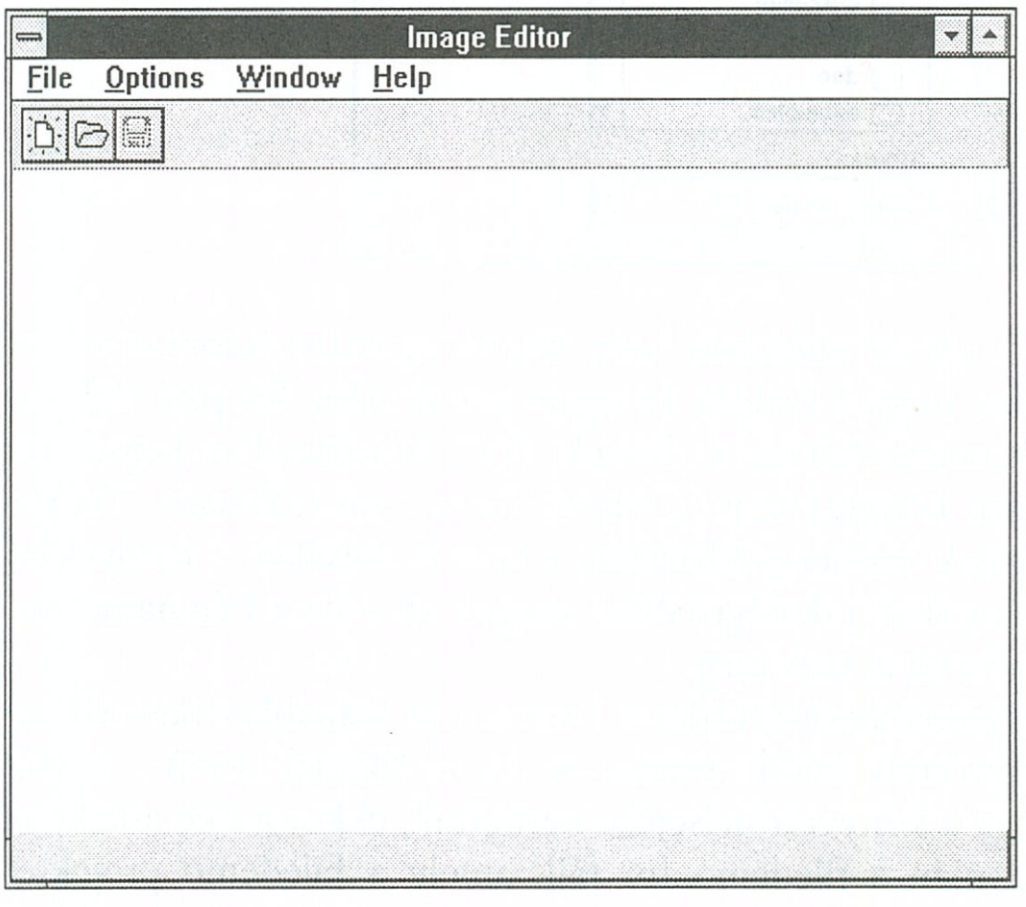
A dialógusablak többi oldalának részei választási lehetőségeket felsoroló listát tartalmaznak. Például az Editor oldalon található Editor Speedsetting listaablakból kiválaszthatjuk, hogy melyik billentyűkombi-

náció milyen szerkesztési funkciónak feleljen meg. Itt csak négy beállítás közül választhatunk: default (normál Windows szerkesztési módszerek), IDE Classic (klasszikus: TurboPascal-WordStar), Brief vagy Epsilon.

■ *A Tools menü*

A Tools menü hozzáférést biztosít a Delphi adott verziójához mellékelt eszközökhöz. Többek között az Image Editorhoz. Az Image Editor (képszerkesztő) átfogó eszköztárának segítségével olyan ikonokat és más képeket készíthetünk ill. változtathatunk meg, amelyeket programjainkban használhatunk. Bejelentkezés után az Image Editor képernyője a 3-3. ábrának megfelelően néz ki. Mint minden „valamirevaló" Windows alkalmazásnak, ennek is saját kiterjedt súgórendszere van.

A többi külső eszköz között találhatjuk a Database Desktopot, amelyre adatbázis-kezelők fejlesztésekor lehet szükségünk, a ReportSmith-t, amellyel jelentést szerkeszthetünk, és egy adatbázismotor (BDE=Borland Database Engine) paraméterezőt (BDE Config).

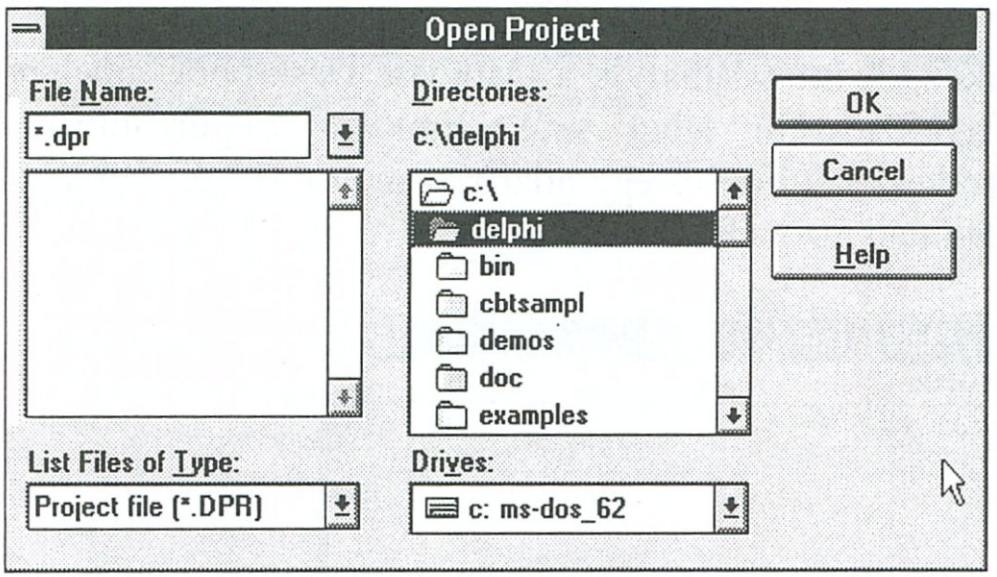


3-3. ábra. Az Image Editor

■ *Programok betöltése és futtatása*

A Delphit nagyon sok és érdekes mintaprogrammal látták el. Fejezzük be ezt a fejezetet annak a folyamatnak az ismertetésével, amely a meglévő Delphi projektek futtatásához szükséges. Ehhez a Filedemo programot fogjuk felhasználni. Ez a demó illusztrálja, hogyan lehet összekötni különböző fájlkezelő-komponenseket. (A fájlkezelő-technikákat a 10. fejezet tárgyalja.)

Ha a File menü Open pontját választjuk, vagy az Open Project ikonra kattintunk az eszközpalettán (az első ikon a palettán), akkor az itt látható Open dialógusablak jelenik meg (előtte lehet, hogy megjelenik egy dialógusablak, amellyel az eddigi munkánkat menthetjük el):



Vegyük észre, hogy az ablakban nem látunk egyetlen állománynevet sem. Ennek az oka, hogy a Delphi az egyes projekteket felépítő fájlokat egy külön .DPR kiterjesztésű (ez a „Projekt fájl") állományban jegyzi, ám az ablakban látható könyvtárban (\Delphi) ilyen nem található. Természetesen lehetőségünk van megváltoztatni a kiterjesztéseket – amelyekkel rendelkező állományokat a dialógusablak megjelenít – ha a File name mezőben új állományfajtát adunk meg.

Lépjünk be a **demos** alkönyvtárba a Directories listaablak használatával. Ennek alkönyvtárai közül válasszuk ki a **filectrl** nevűt, kettőt klikkeljünk rá, hogy a könyvtár belsejébe kerüljünk, ekkor a File ablakban meg kell hogy jelenjen a filedemo.dpr fájl, amely a Filedemo projektet alkotó állományok nevét és elérési útját tartalmazza. Kettőt kattintva

erre, a Delphi beolvassa a Filedemo projektet (várhatóan egy dialógusablak felbukkanásával félbeszakítva, amely megkérdezi, hogy elmentse-e a jelenlegi munkánkat). Ez a projekt is jól mutatja azokat a lehetőségeket, amelyek hamarosan az olvasó kisujjában lesznek. Az F9 lenyomásával elindíthatjuk a programot. (Kilépni, mint minden futó projektből, az ALT F4 billentyűkombinációval lehet.)

4. FEJEZET

Formok tervezése: komponensek, menük, események

Ez a fejezet bemutatja, hogyan alakíthatjuk igényeink szerint formjainkat és az azokat alkotó komponenseket. Egy form vagy egy komponens megjelenését a tulajdonságainak megváltoztatásán keresztül módosíthatjuk. A tulajdonságokat leggyakrabban az objektumfelügyelőben állítjuk be, ezért e fejezet az objektumfelügyelő kezelésének bemutatásával kezdődik. Nem feledkezünk meg a billentyűzetről és az egérről sem. Már az első fejezetben láthattuk, hogy léteznek olyan technikák, amelyek az egeret használják a komponensek (és formok) tulajdonságainak megváltoztatására. Ezen technikák közül szeretnénk itt bemutatni néhányat a billentyűkezelési módszerek mellett. (Megjegyzés: a komponensek pozicionálását, átméretezését és áthelyezését általában egérrel végezzük, kivétel, ha nagyon finom változtatásokról van szó.) Ezután áttekintjük a komponenseket, és bemutatjuk (a teljesség igénye nélkül) a hozzájuk tartozó tulajdonságokat és eseményeket. Ugyancsak itt lesz szó arról, hogyan tervezhetünk menüket a Delphi beépített Menu Designer-ének segítségével, és hogyan menthetjük el a kívánt állományokat ASCII formátumban, hogy azokat egy egyszerű szövegszerkesztőben is meg tudjuk tekinteni ill. ki tudjuk nyomtatni.

Megjegyzés: *A Delphi – verziótól függően – akár száz komponenst is tartalmazhat, ami együttvéve több, mint ezer tulajdonságot jelent. Ezért itt nem lehet célunk a teljesség. A hiányt a beépített help segítségével pótolhatjuk, amelyben további információkat találhatunk a formokról, komponensekről, valamint az ezekhez tartozó tulajdonságokról és eseményekről.*

Lehet, hogy nem egészen precíz, de mindenesetre segít, ha úgy gondolunk a látható komponensek tulajdonságaira, mint a komponens megjelenését befolyásoló dolgokra. A tulajdonságokat fejlesztési és futási időben is megváltoztathatjuk. Például, ha egy formot az egérrel átméretezünk vagy elmozgatunk – ahogy azt az első fejezetben is tettük –, akkor közvetett módon a Left, Top, Width és Height tulajdonságokon változtatunk.

Megjegyzés: *Ha futási időben változtatunk a tulajdonságokon, a szintaxis a következőképpen változik:*

```
object.property: = new value
```

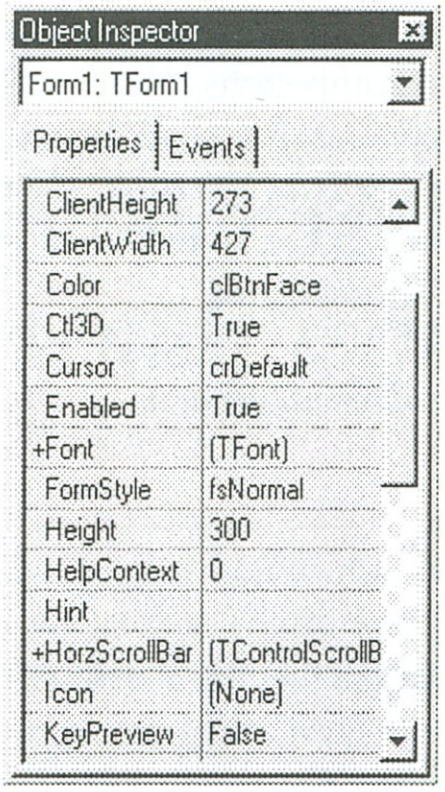
(A tulajdonságokkal kapcsolatos programozási részletek az 5. fejezetben találhatók.)

Az egyik módja annak, hogy lerövidítsük a Delphi (bármelyik verziójának) tanulását, ha megjegyezzük, hogy a tulajdonságok és események nevei ugyanazok maradnak minden komponens esetében. Az ugyanolyan nevű, de különböző komponensekhez tartozó tulajdonságok legtöbb esetben nagyon hasonló szerepet töltenek be. Minden komponens (és form) rendelkezik olyan tulajdonságokkal, amelyek a komponens (és form) helyéért vagy az egy-két szavas eligazító címkék megjelenítéséért (ha a kurzort a komponensre állítjuk) felelősek. Hasonló igaz az eseményekre is. Lényegében ugyanazt az eseményt használjuk, amikor azt szeretnénk, hogy a form reagáljon a felhasználó kattintására, mint amikor azt akarjuk, hogy a form egy nyomógomra történő klikkelésre válaszoljon.

■ *Az objektumfelügyelő használata*

Számos módja van a tulajdonságok beállításának az objektumfelügyelőn keresztül. (A dokumentáció ezeket tulajdonságszerkesztőknek (property editors) hívja.

Az alábbi ábrán a Delphi 32 bites verziójának objektumfelügyelője látható. Fontos tudnunk, hogy az objektumfelügyelőnek két oldala van: az egyik a tulajdonságok beállítására, a másik az események szabályozására szolgál.



(Az objektumfelügyelő 16 bites változata is majdnem így néz ki. Az egyedüli különbség, hogy a fülek alul vannak és nem felül.)

A következő részben ismertetjük a különböző módszereket, amelyekkel a tulajdonságokat lehet beállítani az objektumfelügyelőben. (Ezek mindkét verzióban ugyanazok.)

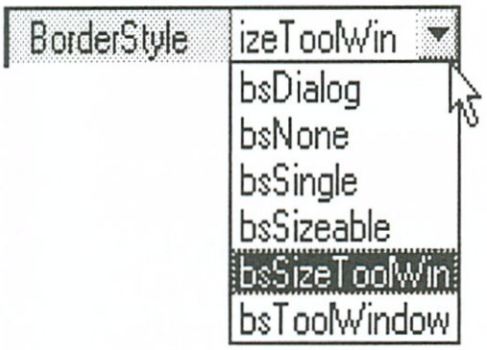
Az egyszerű szerkesztő

Ez a típusú szerkesztő úgy aktivizálódik, hogy a jobb oldali (vagy érték) oszlopra kattintunk, ekkor a kívánt szöveget vagy értéket egyszerűen be lehet gépelni. Ezekben a cellákban lehetőség van a szabvány Windows szerkesztési technikák – úgymint kivágás, beillesztés – használatára. Egy nyilvánvaló példa erre a form Caption (fejléc) tulajdonság beállítása:

A Delphi ellenőrzi, hogy a beírt szövegnek van-e értelme, ha nincs, megjelenik egy „Invalid property value" szöveget tartalmazó üzenetablak (és a hibás érték helyére az eredeti érték kerül). Például a Delphi nem engedi, hogy szöveget írjunk egy cellába, ha oda számot vár.

Szerkesztés legördülő listával

Ezt a szerkesztési módszert a Delphi akkor használja, amikor véges sok választási lehetőség van az adott tulajdonság értékének kiválasztására. Jó példa erre a BorderStyle tulajdonság, amelyik az adott form keretének típusát jellemzi. Az alábbi illusztráció ezt a legördülő listát használó beállítási módszert mutatja be.

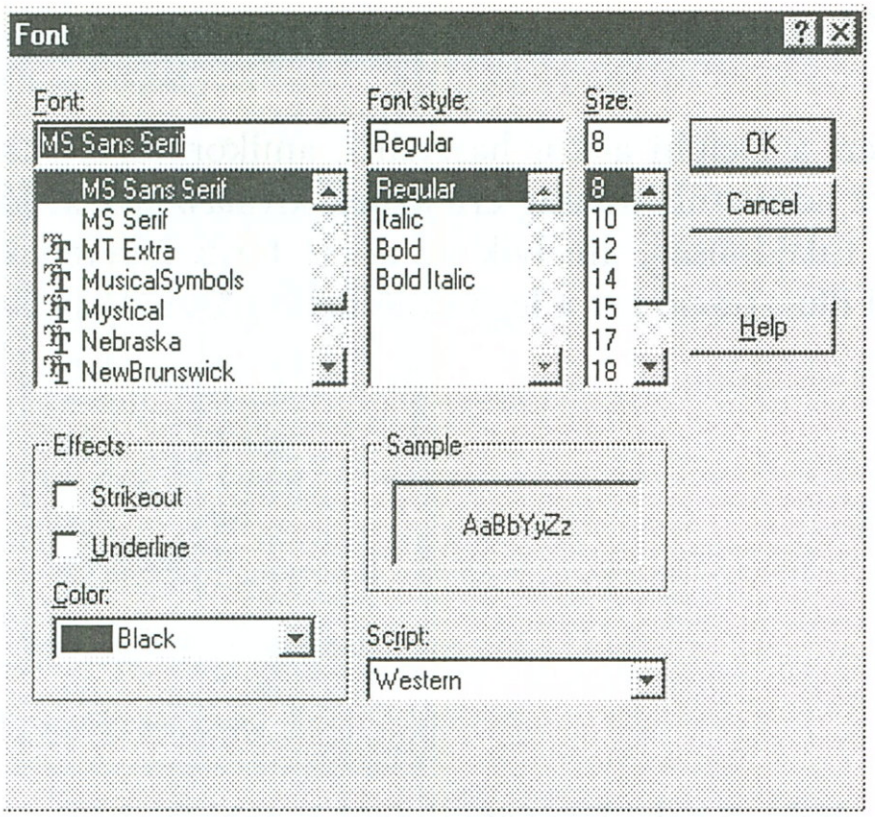


A legördülő listán lévő elérhető tulajdonságok közül is többféleképpen választhatunk ebben a szerkesztőtípusban.

- Ha kettőt kattintunk a cella jobb oldalán lévő nyílra (vagy az ALT - LEFELE-NYÍL kombinációt használjuk), a lista kinyílik. Válasszuk ki a kívánt értéket a listáról.
- Kattintsunk kettőt az értékcellára, ekkor a benne lévő érték megváltozik. Ismételjük ezt addig, amíg meg nem jelenik a kívánt érték.

A dialógusablak-szerkesztő

A három pont (pont-pont-pont) a Delphiben is egy dialógusablak elérhetőségét jelenti (a Windows-szabványnak megfelelően). A felsorolásjel az érték oszlopra történő kattintás után lesz látható. Ha ekkor rákattintunk a három pontra, akkor megjelenik a dialógusablak. (Ugyanez történik, ha kettőt kattintunk az értékoszlopra vagy a CTRL-ENTER billentyűkombinációt használjuk.) Dialógusablakon keresztül szerkeszthető a Font tulajdonság (l. alábbi ábra).



Ha a beállításokat megtettük, akkor kattintsunk az OK nyomógombra. A dialógusablak bezáródik, és a beállítások életbe lépnek.

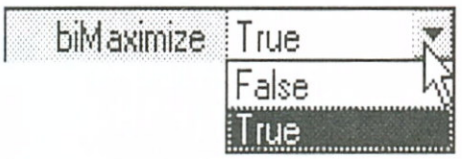
Beágyazott tulajdonságszerkesztő

Ezzel a szerkesztőtípussal olyan tulajdonság beállításánál találkozhatuk, amely több altulajdonságból áll. Ilyen például a BorderIcons tulajdonság (-csoport), amely azért felelős, hogy a címsorban milyen ikonok jelenjenek meg. Háromféle ikon közül választhatunk: a rendszermenü, a maximalizáló gomb és a minimalizáló gomb közül (biSystemMenu, biMinimize és biMaximize). A tulajdonságnév mellett szereplő (+) jel mutatja, hogy egy egész tulajdonságcsoportot állíthatunk be. Kattintsunk kettőt a bal oldali cellára, és a tulajdonságokat tartalmazó lista megjelenik. Az alább látható ábra ezt a szerkesztőtípust mutatja.

- BorderIcons	ize,biMaximize
biSystemMe	True
biMinimize	True
biMaximize	True
biHelp	False

Például ha azt szeretnénk, hogy a formunkon ne jelenjen meg a maximalizáló gomb, akkor a következőket kell tennünk:

1. Kattintsunk kettőt a BorderIcons tulajdonság bal oldali cellájára.
2. Állítsuk a biMaximize celláját hamisra (False) úgy, hogy kettőt kattintunk a jobb oldali cellára, majd válasszuk a False pontot az alább is látható legördülő listáról:

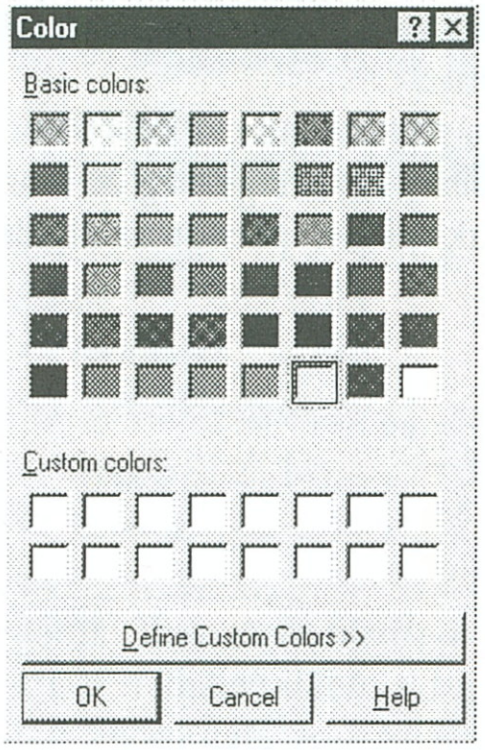


A BorderIcons tulajdonság jó példa azokra a tulajdonságokra, amely valójában értékek egy halmaza. Ezért vannak a jobb oldali cellában látható (vesszővel elválasztott) értékek szögletes zárójelben. A Delphibeli halmazokról többet a 6. fejezetben tudhatunk meg.

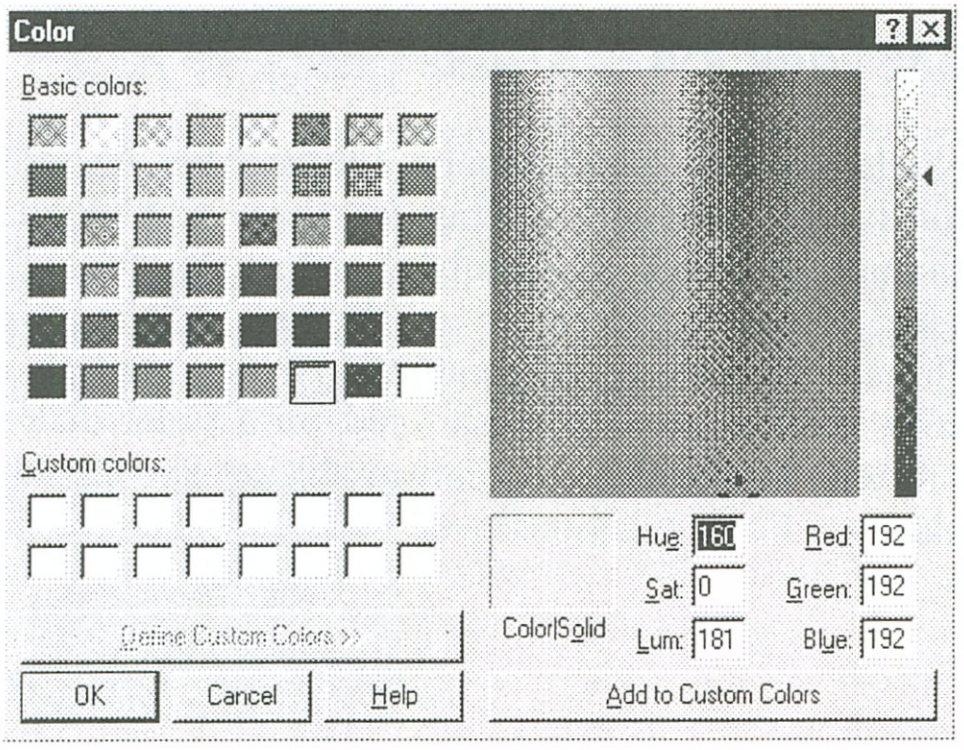
Tipp: *Látható, hogy a Font tulajdonság mellett is ott van a (+) jelzés, amely azt jelzi, hogy itt is az értékek egy halmazát kell beállítanunk. Ha a bal oldali cellára kettőt kattintunk, akkor a dialógusablak helyett a beágyazott tulajdonságszerkesztőt használhatjuk.*

A Color tulajdonság beállítása

Ez a tulajdonság szokatlan módon kétféleképpen állítható be. Az első ezek közül a legördülő listáról történő választás, itt olyan szimbolikus konstansokat választhatunk, amelyek a különböző színeknek felelnek meg. (Olyan konstansok is szerepelnek a listán, amelyek a Windows Control Paneljén beállítható szabványos színkombinációknak – mint pl. a „highlight" beállításnak – felelnek meg.) A másik módszer a Color dialógusablakon keresztül történő színbeállítás. A dialógusablakot (l. 94. old. fent) úgy hozhatjuk elő, ha a Color tulajdonság sorának jobb oldali celláját kettőt kattintunk.



A dialógusablakban a látott színek közül választhatunk, vagy az üres mezők egyikére majd a Define Custom Colors gombra történő kattintás után magunk keverhetjük ki a kívánt színt. Ezt a vörös, a zöld és a kék színek (mindegyik jelenlétének erősségét egy 0 és 255 közötti szám jelzi) megfelelő arányú felhasználásával valamint a Hue (árnyalat), a Saturation (Sat – telítettség) és a Luminosity (Lum – fényesség) jellemzők beállításával tehetjük meg (l. alábbi ábra):



A beállított színnek egy hexadecimális szám felel meg az objektumfelügyelőben.

Tipp: *Ha a színpaletta belsejére kattintunk, akkor a kattintás helyén lévő színnek megfelelően a Delphi automatikusan beállítja a Hue és Saturation értékeket.*

■ *Formok*

A Delphi mindkét verziójában a formokat ugyanazzal az objektumfelügyelőben beállítható (fejlesztési időben) 36 tulajdonsággal lehet leírni. A formok mindkét verzióban ugyanarra a 20 eseményre válaszolhatnak. A most következő néhány fejezetrész célja, hogy az olvasó megismerje a legfontosabbakat ezen tulajdonságok és események közül. Valamivel több időt fogunk tölteni a formokhoz kötődő tulajdonságokkal és eseményekkel, mint a fejezet későbbi részében a komponensekhez kötődőkkel. Ennek oka az, hogy sok tulajdonság és esemény funkciója nem változik a különböző komponensek esetében sem.

Formok tulajdonságai

Alábbiakban röviden ismertetjük a formokhoz tartozó fontosabb tulajdonságokat. (A 12. fejezetben a grafikához kapcsolódó tulajdonságokat bővebben tárgyaljuk.)

Megjegyzés: *Ha egy formhoz kapcsolódó tulajdonságot itt nem találunk meg, akkor használjuk a beépített súgót. Célszerű kihasználni a Delphi helyzetérzékeny súgóját: válasszuk ki a tulajdonságot a kurzor segítségével és nyomjuk meg az F1-et.*

ActiveControl. Ezt a tulajdonságot akkor használjuk, ha egy komponenst illesztünk a formba. Segítségével meghatározhatjuk, hogy kezdetben melyik komponensen legyen a fókusz, amikor a form láthatóvá válik. Ha például a fókusz egy egyszerű nyomógombon van, akkor azt a gombot

körülvevő keret jelzi. Az ActiveControl tulajdonságot gyakran futás közben használjuk. Használatával megállapíthatjuk, hogy melyik komponensen van a fókusz, és hozzáférhetünk annak tulajdonságaihoz (a 7. fejezetben részletezendő objektumkonverziós módszerrel – *casting*).

Auto Scroll. Ha nem változtatunk az AutoScroll tulajdonság True (igaz) alapbeállításán, akkor futási időben a Delphi automatikusan „felszereli" formunkat egy görgetősávval (scrollbar), ha a formon nem fér el az összes vizuális komponens.

BorderIcons. Azt határozza meg, hogy a formon látszódjon-e a szabvány Windows rendszermenü, valamint a minimalizáló és maximalizáló gombok. (L. a korábbi. „Beágyazott tulajdonságszerkesztő" fejezetrészt.)

BorderStyle. Ez a tulajdonság a keret stílusát határozza meg és befolyásolja, hogy mely keret-ikonok (border icons) jelennek meg. (L. még a korábbi „Beágyazott tulajdonságszerkesztő" fejezetrészt.)

Caption. Ennek a tulajdonságnak a megváltoztatását már az 1. fejezetben láthattuk. Ez határozza meg, hogy milyen szöveg látható az adott form fejlécében.

ClientHeight, ClientWidth. Ezek a tulajdonságok a form használható részének (a kereten belüli) szélességéért és magasságáért felelősek. Mindkettőt pixelben (Picture element=képpont=a képernyőn vagy printeren a legkisebb megjeleníthető egység) kell megadni.

Color. Ez a tulajdonság a form háttérszínét határozza meg. A Color tulajdonság különböző színkódjainak beállítására a Delphi vagy legördülő listás szerkesztést használ, vagy – ha a jobb oldali (érték-)cellára kettőt kattintunk – egy dialógusablakban választhatjuk ki a színeket.

Cursor. Ez a tulajdonság a kurzor megjelenéséért felelős (ha az a formon van). A Delphi 16-ban 15, a Delphi 32-ben 16 és 17 kurzorforma közül választhatunk. Az egyszerű nyíltól kezdve, az óraüvegen keresztül az egzotikusabb formákig, minden megtalálható ezek között. Az objektumfelügyelő értékcellájába írható konstansok, mint a cr UpArrow (felfele nyíl)

utalnak a hozzájuk tartozó kurzor alakjára (l. a beépített súgót az elérhető kurzorformák megismeréséhez).

Megjegyzés: A Windows API LoadCursor függvényével lehetőségünk van más kurzorok használatára is. A beépített helpben keressük meg a TScreen objektum Cursors tulajdonságáról írottakat, ha további információkhoz akarunk jutni.

Enabled. Ha ennek a tulajdonságnak megváltoztatjuk True (igaz) alapbeállítását, akkor a form többet nem reagál a külső eseményekre. Általában ezt a tulajdonságot futási időben változtatjuk.

Font. Ez a tulajdonság lehetővé teszi a formon látható szöveg fontjainak futási időben történő beállítását a Canvas tulajdonságon keresztül (l. 5. fejezet). Ez felelős a formra elhelyezett komponensek szövegében használt font megjelenítési módjáért. (A Delphi 16-ban az alapértelmezett font a Windows rendszerfont, a Delphi 32-ben pedig az MS Sans Serif.)

FormStyle. Négy különböző beállítása van ennek a tulajdonságnak. Alapértelmezésben értéke fsNormal, ami azt jelenti, hogy a form egy szabványos ablak lesz. Ha azt szeretnénk, hogy formunk a projektben használt többi form felett helyezkedjen el, akkor válasszuk az fsStayOnTop értéket. A maradék két értéket – fsMDIChild és fsMDIForm – MDI formok esetén használjuk (l. 13. fejezet).

Height, Width. Ezek a tulajdonságok határozzák meg formunk – beleértve a kereteket is – magasságát és szélességét. pixelben (ellentétben a ClientHeight, ClientWidth tulajdonságokkal, amelyek csak a belső tartomány méreteiért felelősek).

HelpContext. Ezt a tulajdonságot akkor használjuk, ha alkalmazásunkba Windows-kompatíbilis súgórendszert szeretnénk beépíteni (l. 13. fejezet).

Hint, ShowHint. A Hint tulajdonság a súgócímkékben megjelenő szövegért felelős, a TApplication objektum OnHint eseményével karöltve. A

ShowHint tulajdonságot True-ra (igaz) kell állítanunk ahhoz, hogy ez az esemény aktivizálódjon.

HorzScrollBar, VertScrollBar. Ezek a tulajdonságok határozzák meg azt, hogy a formban van-e vízszintes ill. függőleges görgetősáv. Ezek beágyazott tulajdonságok, így a bal oldali cellába történő kettős kattintással megjelennek listába szedve az altulajdonságok, amelyeken a beállításokat végrehajthatjuk. A 4-1. táblázatban gyűjtöttük össze és jellemeztük a görgetősávok öt altulajdonságát.

A görgetősávok láthatóvá tételéhez a következőket kell biztosítanunk:

- A Visible tulajdonságot állítsuk True-ra.
- A görgetősáv Range tulajdonsága nagyobb legyen, mint a ClientWidth tulajdonság – vízszintes – vagy mint a Height tulajdonság – függőleges görgetősáv esetén.

4-1. táblázat. A görgetősávok beágyazott tulajdonságai

Tulajdonság	*Funkció*
Increment	Azt határozza meg, hogy a nyilak valamelyikének lenyomásakor milyen mértékben görögjön a képernyő.
Margin	Segítségével megadhatjuk, hogy hány pixelnyire legyen a komponens a görgető sáv szélétől.
Position	Meghatározza a görgetősávon mozgó csúszka helyzetét.
Range	Egy egész típusú tulajdonság, értéke meghatározza, hogy meddig lehet görgetni a képernyőt.
Tracking	Ritkán használt Delphi 32 tulajdonság.
Visible	A görgetősáv láthatóságát határozza meg.

Icon. Ez a tulajdonság határozza meg azt a programunkban használt ikont, amelyik a form minimalizálásakor lesz látható. (Az alkalmazásunk ikonját a Delphi 16-ban az Options | Project | Application oldalon, a Delphi 32-ben pedig a Project | Options | Application oldalon választhatjuk ki.)

KeyPreview. Ennek a tulajdonságnak az alapértéke False (hamis). Ha True-ra (igaz) állítjuk, a billentyűleütések nagy részét először a form Key eseményei dolgozzák fel ahelyett, hogy az a komponens tenné ezt, ame-

lyen éppen az inputfókusz van. (A Key eseményekről bővebben a 6. fejezetben lesz szó.)

Left, Top. Ezek a tulajdonságok határozzák meg a form helyét a képernyő bal felső sarkához képest. Az értékek pixelben értendők.

Menu. Segítségével jelölhetjük ki, hogy melyik menü jelenjen meg a formon. (L. később a „Menütervezés" alfejezetet.)

Name. Ez a tulajdonság minden Delphibeli objektum számára elérhető. A Name tulajdonsággal adhatunk értelmes neveket objektumainknak, ezzel tehetjük programunkat áttekinthetőbbé. Ha itt nem adunk meg nevet, akkor a Delphi az első formhoz a Form1 nevet, a másodikhoz a Form2-t stb. rendeli.

Megjegyzés: *Az objektumok elnevezése körül heves vita bontakozott ki. Néhányan az objektumra utaló előtaghoz kapcsolt értelmes szavakat tartották elfogadhatónak. Erre példák a frmAbout, a frmInitial és a btnHelp. Mások az értelmes szavakból összerakott formák mellett tették le a voksot: AboutForm, FormAbout, vagy HelpButton. Nem szeretnék vitába keveredni, de fontosnak tartom megjegyezni, hogy a jelentéssel bíró nevek használata megkönnyíti a hibakeresést – a konvenciók követése mindig hasznos.*

Az objektumok elnevezésének szabályai:

- Ne használjunk fenntartott szavakat és standard direktívákat (l. a beépített help-et ezek felsorolásáért).
- Az objektum neve bármilyen hosszú lehet, de csak az első 63 karaktert használja a Delphi az azonosításra.
- Az első karakter betű vagy aláhúzás kell hogy legyen.
- A név többi része betűk, számok és aláhúzások bármilyen kombinációja lehet.
- A betűk nagy vagy kicsi volta lényegtelen.

(Ezek gyakorlatilag a Delphibeli *azonosítók* [identifiers] képzésére vonatkozó szabályok, ezek a szabályok vonatkoznak sok Delphi objektumra. Az azonosítókról többet. az 5. és 6. fejezetben olvashatunk.)

Tipp: Ha a Name tulajdonságot megváltoztatjuk, majd a Caption tulajdonság jobb oldali cellájára kattintunk, akkor a Caption tulajdonság értéke felveszi a Name tulajdonságét, amennyiben a két tulajdonság eredetileg is megegyezett.

PixelsPerInch, Scaled. Ha a Scaled tulajdonság True-ra van állítva, akkor a PixelsPerInch tulajdonság meghatározza, hogy a form egy hüvelykjében hány pixel legyen. Ezt a két tulajdonságot főleg futási időben használjuk a Screen objektum PixelsPerInch tulajdonságával együtt. A két tulajdonság kombinációjával meg tudjuk állapítani, hogy a Windows éppen hány pixel/hüvelykes felbontást használ. Ennek ismeretében könnyen írhatunk olyan programot, amely függetleníti a formokat a képernyő-felbontástól.

PopupMenu. Ebben a tulajdonságban adjuk meg annak a felbukkanó-menünek nevét, amely akkor jelenik meg, ha a jobb oldali egérgombbal a form egy üres részére kattintunk. (L. később a „Menütervezés" fejezetrészt, amelyből megtudhatjuk, hogyan készíthetünk ilyen menüket.)

Position. Ezzel a tulajdonsággal könnyen megadhatjuk formunk helyét. Öt lehetőségünk van. A két legfontosabb közül az első, a poDesigned (alapbeállítás), amelyik a Formot azon a helyen hagyja, ahol az tervezéskor volt. A második a poScreenCenter, amelyik automatikusan a képernyő közepére helyezi a formot. A poDefault a Delphire hagyja a form méretének és helyének kijelölését. A poDefaultPosOnly beállításnál a méret nem változik, de az elhelyezést ekkor is a Delphi végzi. A poDefaultSizeOnly választásakor a hely nem változik a tervezéskor beállítotthoz képest, de a méretet a Delphi állítja be.

Print Scale. Ez a tulajdonság leegyszerűsíti a formon lévő információk kinyomtatását. Alapbeállításként a poNone szerepel, ami azt jelenti, hogy léptékválasztás nem történik, és a papíron megjelenő form különbözni fog a képernyőn láthatótól. Ha a poPropotional beállítást választjuk, akkor a Delphi automatikusan úgy választja a léptéket, hogy a kinyomtatott és a képernyőn megjelenített form méretei megegyezzenek. A poPrintToFit érték választásával nyomtatáskor az arányok meg-

maradnak, de a méretek úgy csökkennek, hogy a form ráférjen a nyomtatandó papírra.

Tag. Minden form és komponens rendelkezik ezzel a tulajdonsággal. A Tag értékének egy egész számot adhatunk meg, amellyel információkat tárolhatunk a különben nem elérhető formról vagy komponensről.

Visible. Ez a tulajdonság meghatározza, hogy egy form látható-e futási idő alatt.

WindowState. Ez a tulajdonság három értéket vehet fel. Az alapbeállítás a wsNormal, amely mellett a formunk nem minimalizált és nem is maximalizált. Általában a programban változtatjuk meg a WindowState tulajdonságot a következő értékek valamelyikére: wsMaximized-ra vagy wsMinimized-ra, annak ellenére, hogy ezt természetesen fejlesztési idő alatt is megtehetjük.

Formesemények

Az események felismerése a Delphi ütőerejének alapja, de semmi sem fog történni, amíg meg nem írjuk a megfelelő eseménykezelő kódját. Az eseménykezelő megírásának első lépcsőjeként létre kell hozni a megfelelő eseménykezelőeljárás-sablont az eseménynév jobb oldali cellájára történő kettős kattintással.

Megjegyzés: *Ha kettőt kattintunk egy komponensre vagy formra, a Delphi megjeleníti ezen objektum alapértelmezésű eseménykezelő eljárás sablonját. (Formok esetében ez az OnCreate esemény.)*

Itt következik a legfontosabb formesemények rövid leírása.

OnActivate. A Delphi akkor használja ezt az eseménykezelőt, amikor az inputfókusz áttevődik a formra. Egy formot használó projektekben – miként azt az 1. fejezetben láttuk – ez automatikusan végrehajtódik az alkalmazás lefuttatásakor.

OnClick, OnDblClick. Ezt az eseményt az váltja ki, ha a felhasználó a form egy üres területére kattint (vagy kettőt kattint).

Megjegyzés: *Az OnClick esemény következik be mindig először, még akkor is, ha a felhasználó kettőt kattint.*

OnCloseQuery, OnClose. Az OnCloseQuery eseményt a felhasználó váltja ki, amikor megpróbálja bezárni a formot (pl. kettőt kattint a form bal felső sarkában lévő rendszermenü ikonjára). Ezt az eseményt használhatjuk memóriafelszabadításra vagy akár a form bezárásának megakadályozására. Az OnClose esemény – ugyanakkor – az OnCloseQuery esemény után hajtódik végre, de még ekkor is marad egy utolsó lehetőség a form bezárásának letiltására (nem úgy, mint a Visual Basic-ben). Az OnClose esemény Action paramétere négy különböző értéket vehet fel, amelyek a form bezárásának módját határozzák meg. (caNone, caHide, caFree, caMinimize).

OnCreate. Ez az esemény a form első végrehajtásakor következik be. A form és az ahhoz tartozó komponensek kezdeti tulajdonságainak a beállítására használjuk.

OnDeacitivate. Ez az esemény akkor következik be, amikor a felhasználó egy másik alkalmazásra vált (például a Windows Task Manager vagy az ALT-TAB billentyűkombináció segítségével vagy a Delphi 32-ben a TaskBar használatával). Ennek az eseménynek a használatával bármilyen programrészletet végrehajthatunk (mint pl. felszólítást az adott állomány elmentésére) a fókusz áthelyezése előtt.

OnDestroy. Az OnDestroy a Form megszüntetésekor bekövetkező esemény, amely a formhoz rendelt erőforrásokat automatikusan felszabadítja. Ennek segítségével biztosíthatjuk, hogy az általunk lefoglalt memória felszabaduljon (l. 7. fejezet).

OnDragDrop, OnDragOver, OnMouseDown, OnMouseMove, OnMouseUp. Ezeket az eseményeket az egér mozgását figyelő kódok esetén használjuk. (L. 13. fejezet.)

OnHide. Programunk segítségével elrejthetünk formokat (l. 5. fejezet), ilyenkor az OnHide esemény bekövetkezik.

OnKeyDown, OnKeyPress, OnKeyUp. Ezekkel az eseményekkel követhetjük nyomon a felhasználó billentyűleütéseit. (L. 6. fejezet.)

OnPaint. Az ehhez tartozó eljárásban kezeljük azokat az információkat, amelyek az adott form részeinek az újrarajzolásához szükségesek (pl. mert egy időre egyes részeket más ablakok eltakartak). (L. 12. fejezet.)

OnResize. Ez az esemény akkor következik be, amikor a felhasználó újraméretezi a formot. Általában olyan eljárást írunk hozzá, amelyikkel a formon lévő komponenseket áthelyezzük és átméretezzük.

OnShow. Ez az esemény közvetlenül a form megjelenítése előtt következik be. Ezt legtöbbször a több formot tartalmazó projektekben használjuk. (L. 5. fejezet.)

■ *Komponensek*

A komponensek projektjeink építőkövei. Ezekkel az objektumokkal – úgymint a nyomógombok, szerkesztési felületek – válik a Windows alkalmazások kezelése könnyebbé, mint más alkalmazásoké.

Megjegyzés: *A dokumentáció megkülönbözteti egymástól a látható komponenseket és a vezérlőelemeket (controls). A vezérlőelemek azok a komponensek, amelyeket a felhasználó lát.*

A Delphi különbséget tesz az ablakkal ellátott és az ablakkal el nem látott komponensek között. A különbség az, hogy az előbbire helyezhetjük az inputfókuszt, az utóbbira nem. (Azok számára, akik ismerik a Windows API-t: az ablak nélküli vezérlőelemeknek nincs Windows azonosítójuk.)

Megjegyzés: A könyvben a komponens gyűjtőnév utal majd az ablakkal ellátott és az ablak nélküli komponensekre is.

Hasonlóan a formokhoz, a komponensek is rendelkeznek tulajdonságokkal, amelyek beállítására mind fejlesztési időben (az objektumfelügyelőben), mind futási időben (a programon keresztül) lehetőségünk van. A tulajdonságok nagy része (pl. Height, Width, ShowHint és Hint) lényegében ugyanúgy működik komponensek, mint formok esetében.

Először is rövid leírást közlünk a Delphi mindkét változatában egyaránt gyakran előforduló komponensekről. (A 13. fejezetben lesz még szó tipikus Delphi 32 vezérlőelemekről.) Ezután megtanulhatjuk, hogyan illeszthetjük őket formjainkra. Végül a legfontosabb komponenstulajdonságokkal ismerkedhetünk meg.

Megjegyzés: A könyvben nem kerülnek bemutatásra a Delphi 16 komponenspalettájának VBX, Sample, és OCX oldalán található elemek. (Az adatfeldolgozásnál használt komponenseket a 14. fejezetben ismertetjük.)

A leggyakoribb komponensek áttekintése

A mindkét verzióban megtalálható kb. 70 komponens a komponenspaletta különböző oldalain látható. Az oldalak nevei ugyanazok a Delphi mindkét változatában.

Megjegyzés: A Delphit egyszerűen lehet komponensekkel bővíteni. A Delphi 16-hoz pl. olyan Visual Basic vezérlőelemeket adhatunk, amelyek megfelelnek a Visual Basic 1.0 szabványnak. Ez már önmagában többszáz eszközt jelenthet, amelyekkel speciális igényeinket is kielégíthetjük. (Figyelem: Nem ajánlom a VBX-ek beillesztését olyan alkalmazásokba, amelyeket Windows 95 alatt is futtatni szeretnénk, mivel a Microsoft ezek használatát igyekszik kiikatni a Windows 95-ből.) Miután a Delphi 32-höz OCX vezérlőelemeket is adhatunk, így ebben a változatban is komponensek széles skálája áll rendelkezésünkre. Új komponensek telepítésének mikéntjét a help „Registering Components" oldalán találhatjuk.

Visual Basic Tipp: *A Delphi 16 VBX oldala Visual Basic vezérlőelem-példányokat tartalmaz. (A Visual Basic vezérlők VBX kiterjesztésű fájlokban vannak.) Elképzelhető, ha a Visual Basic Professional Edition változatával rendelkezünk, hogy annak komponensei nem lesznek kompatíbilisek a VBX oldalon találhatókkal. A Visual Basic 3.0 GAUGE.VBX vezérlője például nem kompatíbilis a Delphi-belivel, ezért az utóbbinak új nevet kell adnunk, hogy elkerüljük a rendszer esetleges lefagyását. Ugyanez igaz egyes külső vezérlőelemekre is.*

A Standard oldal

A komponenspaletta Standard oldalán találhatók azok a komponensek, amelyekre a programozónak általában leggyakrabban szüksége van. Ezeket a 4-2. táblázatban foglaljuk össze. (Ne felejtsük, hogy a Delphi komponensek nevei mindig nagy T betűvel kezdődnek.)

Az Additional oldal

Az Additonal oldal olyan komponenseket tartalmaz, amelyet speciálisabb feladatok – úgymint fülecskékkel ellátott jegyzetfüzet, vagy a grafikához kapcsolódó gombok – megvalósítására használhatunk. Ezek a komponensek kissé eltérhetnek a különböző Delphi verziókban. A 4-3. táblázatban az Additional oldal komponenseinek rövid jellemzését találhatjuk. Azokat a komponenseket, amelyek nem elérhetőek a Delphi 32 ezen oldalán * jelöli. (A grafikus vonatkozásról a 12. fejezetben olvashatunk.)

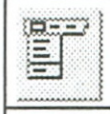
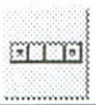
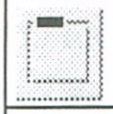
A Dialogs oldal

A komponenspaletta Dialogs oldala megegyezik a Delphi két változatában. Ez az oldal tartalmazza a Windows felhasználók körében jól ismert dialógusablak-mintákat. Ezeket a 4-4. táblázat mutatja be. (l. még a 13. fejezetet.)

A System oldal

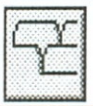
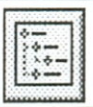
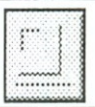
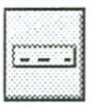
A komponenspaletta System oldalán található komponensek közül a legtöbb megtalálható a Delphi mindkét változatában. A System oldalon vannak azok a komponensek, amelyek segítségével meghajtókat és fájlokat tudunk kezelni (l. 11. fejezet) valamint DDE-t (dinamikus adatcserét) és OLE-t (objektumbeépítést és -hozzászerkesztést) tudunk megvalósítani (l. 13. fejezet).

4-2. táblázat. A komponenspaletta Standard oldala

Ikon	*Név*	*Leírás*
	TMainMenu	A menüstruktúra megtervezéséhez használjuk.
	TPopupMenu	Segítségével felbukkanó menüt hozhatunk létre.
	TLabel	Más komponensek címkézésére és statikus szöveg megjelenítésére szolgál.
	TEdit	*Egy sor* szerkesztésére alkalmas területet hoz létre.
	TMemo	*Több sornyi* szöveg szerkesztésére biztosít területet.
	TButton	Creates command buttons
	TCheckBox	Jelölőablak-komponens két állapotú opciók beállításához
	TRadioButton	Több egymást kölcsönösen kizáró opció létrehozásakor használjuk.
	TListBox	Egy vagy több elemből álló lista létrehozására használjuk.
	TComboBox	A TEdit és a ListBox kombinációja, a felhasználó újabb elemmel bővítheti a listát ill. a meglévők közül választhat.
	TScrollBar	Görgetősáv-komponens, meghatározza, hogy a listának vagy formnak melyik része legyen látható.
	TGroupBox	Csoportablak-komponens, az összetartozó komponenseket szervezi csoportba.
	TRadioGroup	A csoportablak egy változata, amelyben a rádiógomb-csoportot hozhatunk létre.
	TPanel	Más komponensek tárolását valósíthatjuk meg benne. Állapotjelző, gyakran használjuk sor és komponenspaletta készítésére (l. 13. fejezet).

A fontos TTimer komponens is ezen az oldalon található (l. később A „TTimer komponens" alfejezetet). Ezt az oldalt a 4-5. táblázat foglalja össze.

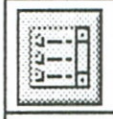
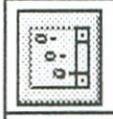
4-3. táblázat. A komponenspaletta Additional oldala

Ikon	*Név*	*Leírás*
	TBitBtn	Billentyűkomponens, amelynek felületére szöveget és grafikát is lehet helyezni.
	TSpeedButton	Billentyűkomponens, amelynek felületén általában grafikát jelenítünk meg. A panelkomponensen használva eszközpalettát készíthetünk.
	TTabSet(*)	Segítségével a 16 bites változatban lapozható dialógusablakokat (ilyen az objektumfelügyelő ablaka is) hozhatunk létre. A TNoteBook komponenssel együtt használva a TTabbedNote-nál nagyobb rugalmasságot nyújtó elemhez jutunk.
	TNotebook(*)	Egy olyen komponens, melynek oldalait tetszés szerint rendezhetjük be.
	TTabbedNote(*)	Egy lapozható, fülekkel ellátott komponens a Delphi 16-ban.
	TMaskEdit	Egy olyan szerkesztésiablak-komponens, amelyben a felhasználó által bevihető adatokat limitálhatjuk (megadhatjuk, az érvényes karaktereket).
	TOutline(*)	Segítségével összefüggő adatok viszonyának hierarchiáját jeleníthetjük meg.
	TStringGrid(*)	Szöveges táblázatot létrehozásához használjuk.
	TDrawGrid	Képek táblázatos megjelenítésére szolgál.
	TImage	Ikonok, bittérképek és Winows metafile-ok megjelenítéséhez használjuk.
	TShape	Különböző alakzatok (körök, ellipszisek, téglalapok és lekerekített sarkú téglalapok) megjelenítésére alkalmas komponens.
	TBevel	Süllyesztett vagy kiemelt egyenes és téglalap elhelyezésére használjuk.
	THeader(*)	A 16 bites Delphi méretezhető komponense, amelynek celláiba különböző fejlécfeliratokat írhatunk. Gyakran táblazatokkal együtt használjuk.
	TScrollBox	Görgethető felület létrehozására használjuk.

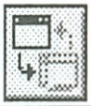
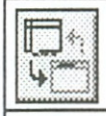
4-4. táblázat. A komponenspaletta Dialogs oldala

Ikon	*Név*	*Leírás*
	TOpenDialog	Az Open File általános Windows dialógusablakot hozza létre állományok megnyitásához.
	TSaveDialog	A Save File általános dialógusablakot hozza létre fájlok elmentéséhez.
	TFontDialog	Fontbeállítások szabványos dialógusablaka.
	TColorDialog	A színbeállításokhoz használt szabványos dialógusablak.
	TPrintDialog	A Print általános dialógusablaknak megfelelő Delphi komponens.
	TPrinterSetupDialog	Az általános Printer Setup dialógusablakot érhetjük el ezen keresztül. (Közvetett módon a Print dialógusablakból is eljuthatunk ide.)
	TFindDialog	A Find (kereső) szabványos dialógusablaknak felel meg.
	TReplaceDialog	A szabványos Replace (helyettesítés) dialógusablakot érhetjük el ezen keresztül.

4-5. táblázat. A komponenspaletta System oldala

Ikon	*Név*	*Leírás*
	TTimer	Időkap-csoló komponens, segítségével megadott időközönként hívhatunk meg alprogramokat.
	TPaintBox	Egy téglalap alakú területet hoz létre, amelyre a program rajzolhat.
	TFileListBox	Egy olyan listaablak, amely az aktuális meghajtó könyvtárrendszerét jeleníti meg.
	TDirectoryListBox	Megjeleníti az aktuális meghajtó könyvtárrendszerének felépítését.
	TDriveComboBox	Meghajtóválasztáshoz használt csatolt listaablak.
	TFilterComboBox	A TFileListBox által megjelenített állományok szűréséhez használt komponens.

4-5. táblázat. A komponenspaletta System oldala (folytatás)

Ikon	*Név*	*Leírás*
	TMediaPlayer	Multimédia-állományok kezelésénél használt eszköz.
	TOLEContainer	OLE kliens komponens.
	TDDEClientConv	DDE kliens létrehozásához szükséges komponens.
	TDDEClientItem	Meghatározza, hogy milyen adatokat küldjön a szerver a kliensnek.
	TDDEServerConv	DDE szerver létrehozásához szükséges komponens.
	TDDEServerItem	Meghatározza a kliensnek elküldendő adatokat.

Komponensek illesztése a formra

Ha már kiválasztottuk, hogy melyik komponenst helyezzük az aktuális formra, akkor annak beillesztésére a következő lehetőségek kínálkoznak:

- Ha egy komponensre kettőt kattintunk, akkor a Delphi azt az aktuális formra helyezi az alapértelmezésnek megfelelő méretben. (Ha a form közepét már elfoglalja egy komponens, akkor az új jobbra alulra kerül.)
- Ha a komponenst egy adott helyre kívánjuk helyezni:
  1. Kattintsunk a komponensre.
  2. Kattintsunk a form azon helyére, ahová a komponens bal felső sarkát kívánjuk helyezni. Komponensünk az alapértelmezésnek megfelelő méretben és a kijelölt helyen fog megjelenni.

Természetesen lehetőség van a komponens méreteinek megváltoztatására. Ennek céljából ahelyett, hogy a bal felső saroknak megfelelő helyre kattintanánk, állítsuk a kurzort a komponens helyére, tartsuk lenyomva a bal oldali egérgombot, és húzzuk az egeret addig, amíg a Delphi által felrajzolt körvonal a kívánt méretű és helyzetű nem lesz, ekkor engedjük el az egérgombot.

Többszörös komponensek beillesztése

Gyakran előfordul, hogy egy bizonyos komponenst többször is fel akarunk használni egy formban. (Például gyakran több parancsgombra vagy szerkesztőablakra van szükségünk.)

A következőket kell tennünk egy komponens többszöri beillesztéséhez:

1. Tartsuk lenyomva a SHIFT gombot.
2. Kattintsunk a komponenspaletta kívánt komponensére. (A kiválasztás után a shiftet már fel lehet engedni.)
3. Kattintsunk a formon azokra a helyekre, ahová a komponens egyes példányainak a bal felső sarkát helyezni kívánjuk (ekkor az alapértelmezésbeli méretben fognak megjelenni), vagy húzzuk az egeret a bal oldali gomb lenyomásával, amíg a kívánt méretet és alakot meg nem kapjuk.

Megjegyzés: *Ha ezt a módszert használjuk a komponensek formra helyezésére, akkor ne felejtsünk el a kurzor ikonra kattintani annak érdekében, hogy ismét a megszokott módon dolgozhassunk a komponensekkel. (A kurzor ikon a komponenspaletta minden oldalán megtalálható, a bal szélen.)*

Tartalmazó (konténer) komponensek

Vannak olyan komponensek, amelyeket csoportosan használunk, ilyenek például a rádiógombok is. Csoportokba szedéssel érhetjük el, hogy a Delphi automatikusan tudja, melyik gombot kell kapcsolnia egy adott gomb lenyomásakor. A Delphi mindkét verziója hat olyan komponenssel rendelkezik a formon kívül, amelyek *tartalmazó* (konténer) komponensként szolgálhatnak (ezekkel különíthetünk el csoportokat). A tartalmazó komponenst alkotó komponensek egyként viselkednek fejlesztési időben (néha ezt szülő-gyerek kapcsolatnak nevezik). Például amikor a szülőkomponenst mozgatjuk, a gyermekkomponensek is mind vele „tartanak". A tartalmazó komponensek a következők: TGroupBox, TRadioGroupBox, TPanel, TNotebook, TTabbedNotebook és TScrollBox.

A szülők létrehozásának legegyszerűbb módja az, hogy a megfelelő komponenst előbb adjuk a formhoz, mint a gyermekkomponenseket. Ha

egy konténerkomponenst helyeztünk a formra, akkor azt feltétlenül jelöljük ki.

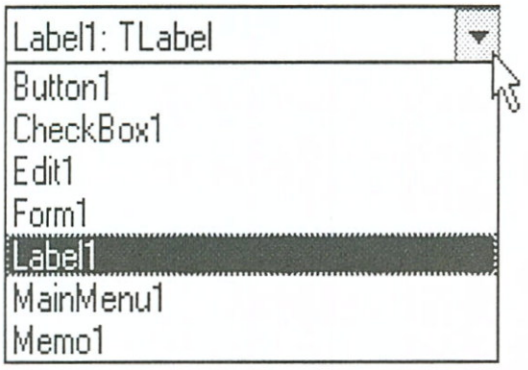
Ezután a megszokott módon (pl. a szülő belsejére történő kattintással) adjunk komponenseket. Például a tartalmazókomponens kiválasztása után ha a kívánt komponensre kettőt kattintunk, akkor az megjelenik (gyermekként) a szülő középső részén.

Megjegyzés: *Az aktív szülőkomponensre több egyforma gyermekkomponenst is elhelyezhetünk a shift+kattintás módszert használva. Ez leginkább eszközpaletták készítésénél hasznos (l. 13. fejezet).*

A formon lévő komponensek manipulálása

Csak a form kijelölt komponenseivel dolgozhatunk. A kijelöléshez a következő módszerek állnak rendelkezésünkre:

- Kattintsunk a komponens belsejébe.
- Nyissuk ki az ojektumokat tartalmazó legördülő listát az objektumfelügyelőben és válasszuk ki a kívánt komponenst. (L. alábbi illusztráció.)



- Ha az inputfókusz a formban van, akkor a TAB segítségével helyezhetjük át a fókuszt komponensről komponensre.

Ha kijelölünk egy komponenst a formon, akkor annak kerületén kis fekete négyzetek, ún. méretezőfogantyúk jelennek meg. Ezek húzásával a komponens átméretezhető (l. később a „Vezérlőkomponensek átméretezése és átformálása" alfejezetet.)

Több komponens kiválasztása

Gyakran van szükségünk arra, hogy egyszerre több komponenssel dolgozzunk (pl. amikor egymáshoz igazítjuk őket). Több komponens kiválasztásának legegyszerűbb módját a következő lépések mutatják:

1. Tartsuk a SHIFT gombot lenyomva.
2. Kattintsunk a kiválasztandó komponensekre.

Egy másik, néha igen hasznos módszer:

1. Képzeljünk el egy téglalapot, amelyik csak azokat a komponenseket veszi körbe, amelyeket ki szeretnénk választani. Helyezzük a kurzort e képzeletbeli téglalap egyik csúcsára, és nyomjuk le a bal oldali egérgombot.
2. Miközben lenyomva tartjuk a bal oldali egérgombot, az egér segítségével addig húzzuk a pontozott téglalapot, amíg az nem tartalmaz minden kiválasztani kívánt komponenst. Engedjük fel a gombot.

Függetlenül attól, hogy melyik módszert használjuk a vezérlőelemek egy csoportjának kiválasztására, sikeres kijelölés esetén minden komponensen látszani kell a szürke méretezőfogantyúknak.

Megjegyzés: Ha a komponensek egy szülőhöz (pl. a TPanelhez vagy TGrpupBoxhoz) vannak csatolva, akkor a húzás megkezdése előtt nyomjuk le a CTRL billentyűt.

A formon lévő összes komponens kiválasztására is van lehetőségünk az Edit | Select All pont választásával. Egy komponenscsoport kijelölését megszüntethetjük, ha a form egy üres részére kattintunk.

Komponensek mozgatása

Ahhoz, hogy egy komponenst áthelyezzünk egy másik helyre a következőket kell csinálnunk:

1. Jelöljük ki a komponenst.
2. Helyezzük a kurzort a vezérlőelem belsejébe, és az egér segítségével (a bal oldali gomb lenyomva tartásával) húzzuk a komponenst az új helyére. (Vigyázzunk, nehogy a méretezőfogantyúkat húzzuk, mert ekkor mozgatás helyett átméretezést végzünk.)

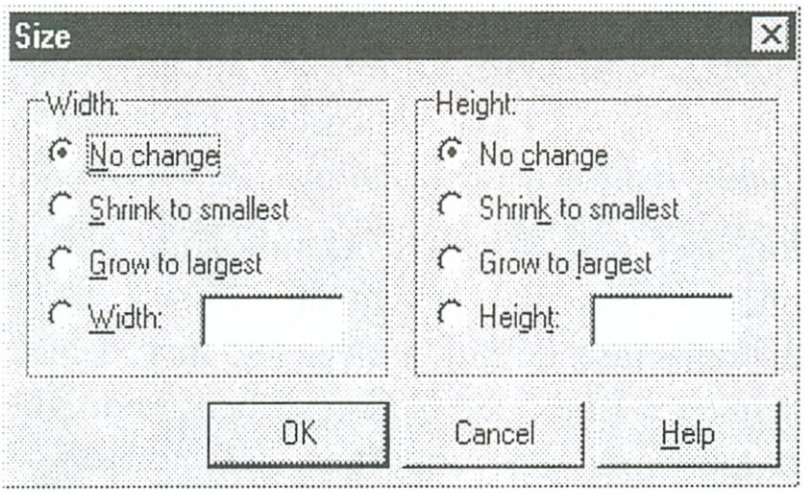
Vegyük észre, hogy a komponensek mozgatáskor elég szakaszosan változtatják helyüket. Ahogy a régi számítógépes vicc mondja, ez nem hiba, ez egy szolgáltatás. Ez azért van, mert a komponensek lehetséges helyeit a rácspontok határozzák meg. A Delphi 32-ben a Tools menü Environment oldalán, a 16 bites verzióban pedig az Options menü ugyanezen oldalán változtathatunk ezen a „sajátosságon". (Úgy nevezik, hogy „Snap to grid") Ebben a dialógusablakban a rácsot finomabbá ill. durvábbá tehetjük.

Megjegyzés: Ha komponensek egy csoportját jelöltük ki, akkor azok áthelyezéskor párhuzamosan mozognak.

Vezérlőelemek méretének és alakjának megváltoztatása

A legegyszerűbben a méretezőfogantyúkkal változtathatjuk meg egy vezérlőelem méretét és alakját. Átméretezése előtt jelöljük ki az adott komponenst. Ekkor láthatóvá válnak a fekete méretezőfogantyúk. A megfelelő fogantyút húzzuk az egérrel a kívánt pozícióba.

A méreteket az Edit | Size pont választásával megjelenő dialógusablakban is beállíthatjuk (l. ábra).



A beállítások után kattintsunk az OK gombra.

Megjegyzés: Több komponens egyidejű átméretezése is hasonlóképpen történik: először jelöljük ki őket, majd nyissuk ki a Size dialógusablakot, és töltsük ki a megfelelő mezőket.

Komponensek törlése

Ha az aktuális formról komponenseket szeretnénk letörölni, akkor először jelöljük ki őket, aztán nyomjuk meg a DEL gombot vagy válasszuk az Edit | Delete pontot.

Tipp: A letörölt komponenseket visszaállíthatjuk az Edit | Undelete pont választással, azonban ez csak akkor működik, ha az Undelete pontot rögtön a törlés után használjuk.

Komponensek kivágása, másolása a vágólapra és beillesztése

A Delphiben lehetőségünk van a komponensek kivágására, vágólapra másolására és beillesztésére. A másolás-beillesztés végezhető két form között vagy egy form és egy szülőkomponens között a szabvány Windows-os módszerekkel (kivágás: CTRL-X, másolás a vágólapra: CTRL-C stb.) Használhatjuk az Edit menü megfelelő pontjait is. Fontos tudni, hogy egy komponens átmásolásakor annak nevét a Delphi megváltoztatja, de tulajdonságait megőrzi. Még fontosabb, hogy a Delphi továbbra is a régi eseménykezelőket köti az új komponensekhez. (A 7. fejezetben lesz szó arról, hogyan dolgozhatunk a több esemény által használt eseménykezelőkkel.)

Komponensek elrendezése

Ha egeret használunk a komponensek elhelyezésére, beleütközünk abba a problémába, amelyet a komponensek széleinek egyvonalba állítása jelent. Amennyiben alkalmazásunknak biztosítani akarjuk a megfelelő megjelenést, akkor fontos, hogy komponenseinket a lehető legtökéletesebben rendezzük el. Miután kiválasztottuk a komponensek egy csoportját, két módszer áll rendelkezésünkre az optimális elrendezés megvalósítására:

- Használjuk az Align ablakot (válasszuk a View | Alignment Palette pontot), és kattintsunk a gombra, amelyik a kívánt módon igazítja el a komponenseket.
- Válasszuk az Edit | Align pontot az Align dialógusablak megnyitásához. Ezután kattintsunk a kívánt mezőkre.

Megjegyzés: Függetlenül attól, hogy melyik módszert használjuk, mindig az első kiválasztott komponens szolgál majd viszonyítási alapként. Például, ha az egy kicsit balra kilóg a többi közül, akkor az Align dialógusablak Left sides pontjának választásával a többi komponens balra fog elmozdulni.

Néhány gyakori komponens áttekintése

Az alfejezet eleje tárgyalja azokat a gyakori komponenstulajdonságokat és eseményeket, amelyek fontosak az inputfókusz manipulálásakor. Utána a gyakoribb komponensekről esik szó. (Nem túlzás azt állítani, hogy 10-15 komponens alkotja a Delphi projektek vizuális felületének zömét.)

Amíg a legfontosabb tulajdonságokról és eseményekről szót ejtünk, amelyek időről időre előfordulnak, nem ismételjük meg azoknak a tulajdonságoknak és eseményeknek, úgymint Color, Cursor, Enabled, Left, Name, Top, Visible, Width, OnClick, OnDblClick stb. tárgyalását, amelyek lényegében úgyanúgy működnek formok és komponensek esetében. Hasonlóan, a súgócímkék (amelyek akkor bukkannak fel, amikor a kurzor egy komponens felett pihen) megjelenítéséért felelős eljárás is ugyanaz komponensekre, mint formokra. (A 6. fejezetben tárgyaljuk a billentyűzethez kapcsolódó eseményeket, a 13. fejezet pedig az egérrel kapcsolatos eseményekről, pl. OnMouseDown-ról vagy az OnDragOver-ről szolgál bővebb információkkal.)

Érdemes itt kiemelnünk az Align tulajdonságot, amely sok komponenshez kötődik. Ehhez a tulajdonsághoz olyan értékek tartoznak, mint alTop, alRight, alBottom és így tovább. Az Align beállításával a komponenseket a keret kívánt oldalához rögzíthetjük.

Megjegyzés: Azok a tulajdonságok, amelyek a vezérlőelemek helyéért felelősek, úgymint Align, Left és Top, mindig a tartalmazókomponens határait veszik alapul.

Tipp: Ha több vezérlőt jelölünk ki, akkor az objektumfelügyelő csak azokat a tulajdonságokat mutatja, amelyekkel mindegyik rendelkezik. Ez megkönnyíti a Height, Width és ezekhez hasonló olyan tulajdonságok beállítását, amelyeken azt szeretnénk, hogy a komponensek osztozzanak.

■ *Fókusszal kapcsolatos tulajdonságok és események*

Egy adott időben vagy a form, vagy egy azon található komponens lehet csak aktív, azaz rendelkezhet az inputfókusszal. Emlékezzünk, egy módszert már láttunk, amellyel azt tudjuk beállítani, hogy kezdetben melyik komponensen legyen a fókusz: ez a form ActiveControl tulajdonságának beállítása volt. Ha nem állítjuk be az ActiveControl tulajdonságot, akkor a Delphi megkeresi azokat a komponenseket, amelyeknek TabStop tulajdonsága True-ra van állítva. Azután a fókuszt az a látható (és enabled-re állított) komponens kapja meg, amelynek a TabOrder tulajdonsága a legalacsonyabb értékkel rendelkezik.

Az OnEnter esemény. Ez az esemény akkor következik be, amikor egy komponens aktívvá válik, vagy ráhelyeződik a fókusz. Ennek segítségével írhatunk olyan kódot, amely az adott komponens aktivizálásakor hajtódik végre.

Az OnExit esemény. Ez az esemény akkor következik be, amikor a fókusz áthelyeződik egy aktív komponensről. Ennek segítségével írhatunk olyan kódot, amely akkor kapja meg a vezérlést, amikor az adott vezérlőelem elveszti a fókuszt.

A Caption tulajdonságok és a gyorsítóbillentyűk. Sok Windows alkalmazás használ gyorsítóbillentyűket a gyors fókuszáthelyezés és nyomógombválasztás érdekében. Ezeket a billentyűket aláhúzás jelzi a feliratokban. Ezután a felhasználó az ALT és az aláhúzott betű billentyűjének együttes lenyomásával aktivizálhatja az adott funkciót.

Egy komponenshez gyorsítóbillentyűt úgy rendelhetünk, hogy a gyorsítóbetű elé egy „&" jelet írunk. Például ha a TButton Caption tulajdonságát &Quitre állítjuk, akkor a felhasználó az egérrel való kattintás helyett az ALT -Q lenyomásával aktivizálhatja a nyomógomb OnClick eseménykezelő eljárását.

A Ctl3D tulajdonság. Ez egy logikai (True/False) tulajdonság, amely azt határozza meg, hogy egy vezérlőelem 3- vagy 2-dimenziós formában jelenjen-e meg. A Delphiben az alapbeállítás True, ami azt jelenti, hogy a komponenseknek mélységük is van.

A TLabel (címke) komponens

Ez a komponens a más vezérlőelemek (pl. szerkesztőablakok) azonosítására szolgál. A TLabel-t közvetlenül a megjelölendő komponens mellett helyezzük el. Használhatjuk még a felhasználó által nem szerkeszthető szöveg megjelenítésére is.

Tipp: *A címkét használjuk akkor is, ha egy olyan komponenshez szeretnénk gyorsítóbillentyűt rendelni, amelyiknek nincs Caption tulajdonsága (pl. szerkesztőablakok). Így amikor a felhasználó lenyomja a gyorsítóbillentyűt, a címke FocusControl tulajdonságának értékét képező vezérlő kapja meg a fókuszt.*

Alignment. Háromféleképpen rendezhetjük a címkén belüli feliratot. Pl. a tulajdonság taCenter-re állításával a Delphi a szöveget a címke közepére rendezi. A másik két lehetséges érték a taLeftJustify (balra rendez) és a taRightJustify (jobbra rendez).

AutoSize. Ha ezt a tulajdonságot True-ra állítjuk, akkor a címke automatikusan hozzáidomul a szöveghez. True-beállítás esetén a változó feliratokat is követik a címke méretei.

Transparent. Grafika (l. 12. fejezet) használata esetén tesz jó szolgálatot. True értékre állításakor a címke átlátszó lesz, és az alatta lévő grafika láthatóvá válik.

Wordwrap. Ezt akkor állítsuk igazra, ha a sor végéről kilógó szavakat a következő sorba szeretnénk helyezni (sortörés).

A TEdit komponens

A Tedit komponens egy egysoros szerkesztési területet biztosít, amelybe a felhasználó szöveges adatokat gépelhet. Alkalmas továbbá a felhasználónak szánt üzenetek megjelenítésére is. A TEdit vezérlőelemek AutoSize tulajdonsága alapértelmezésben True-ra van állítva, ami azt eredményezi, hogy a szerkesztési terület mérete követi az esetleges fontváltoztatásokat (a szélesség nem változik).

AutoSelect. Értékének True-ra állításával elérhető, hogy amikor a szerkesztési terület megkapja a fókuszt, az abban található szöveg kijelölve jelenjen meg. (Alapértéke True.)

CharCase. Háromféle értéket vehet fel. Az ecNormal beállítással a szerkesztési ablakban kis- és nagybetűk egyaránt megjeleníthetők. Az ecLowerCase (ecUpperCase) választással a Delphi automatikusan kisbetűkké (nagybetűkké) alakítja a felhasználó által begépelt vagy az általunk megadott sor nagybetűit (kisbetűit).

HideSelection. Ez a tulajdonság szabályozza azt, hogy a szerkesztési komponensben a kiválasztott szöveg kijelölve maradjon-e a fókusz áthelyezése után is. Ha False-ra állítjuk (az alapérték True), a szöveg kijelölt állapotban marad.

MaxLength. Ez a tulajdonság határozza meg a felhasználó által begépelhető szöveg maximális hosszát. Az alapérték 0, ennél a beállításnál a rendszer kapacitása szab határt a hossznak. Minden más érték korlátozza a karakterek számát.

PasswordChar. Ha el kívánjuk rejteni a felhasználó által begépelt karaktereket, akkor állítsuk értékét #0-tól különböző karakterre.

ReadOnly. A ReadOnly tulajdonság értékét True-ra állítva megakadályozzuk azt, hogy a felhasználó szöveget írhasson a területre. Ez egy gyakori alternatívája a címke használatának rögzített feliratok megjelenítéséhez.

Text. Ez a tulajdonság határozza meg, hogy kezdetben milyen szöveg jelenjen meg a vezérlőelemben. Alapbeállításként a vezérlőelem neve szerepel.

Az OnChange esemény. Ez az esemény akkor következik be, amikor a felhasználó megváltoztatja a szerkesztési területben lévő adatot.

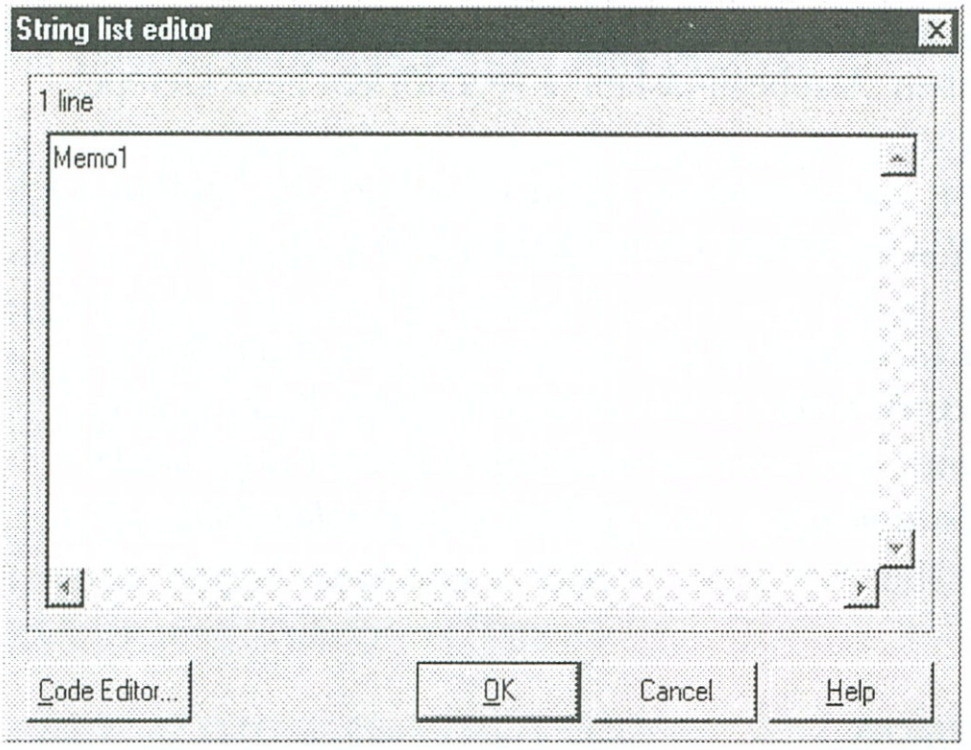
Figyelem: *Ne kapcsoljunk az OnChange eseményhez olyan eljárást, amely a szerkesztési terület tartalmát megváltoztatja, hiszen ez újra indítja az eljárást, ami ismét megváltoztatja a szöveget és így tovább, amíg a program le nem fagy. (Ezt gyakran eseményzuhatagnak nevezzük.)*

A TMemo komponens

A TMemo komponens lehetőséget kínál a többsoros szövegek kezelésére. A TMemo komponens segítségével fájlban tárolt adatok gyors beolvasása is megvalósítható (l. 10. fejezet). TMemo komponensek a Delphi 16-ban 32 K-nyi, a Delphi 32-ben pedig lényegében korlátlan hosszú (1 Gigabyte) szöveget olvashatnak be és tárolhatnak. A TMemo komponens sok tulajdonsága – többek között a MaxLength és a ReadOnly – hasonlóképpen működik, mint a TEdit-é.

Ha a Tmemo komponensre görgetősávot szeretnénk helyezni, akkor ennek megfelelően állítsuk be ScrollBars tulajdonságát. Az Align tulajdonság a megszokott módon használható egy kis különbségtől eltekintve, nevezetesen, hogy az alClient érték választásával a TMemo komponens elfoglalja az egész formot. Most röviden ismertetjük a TMemo komponensre jellemző legfontosabb tulajdonságokat.

Lines. Ez a tulajdonság a komponens tartalmát kezeli. Fejlesztési időben több sorból álló szöveget (minden sor max. 255 karakter hosszú lehet Delphi 16 esetében, míg Delphi 32-ben lényegében nincs felső korlát) illeszthetünk be a jobb oldali cellára történő kettős kattintással. Ennek hatására kinyílik egy dialógusablak, amelybe begépelhetjük a kívánt szövegsorokat.



WantsTabs, WantReturns. A Windows alá írt programokban rendszerint a tabulátorral mozoghatunk a vezérlőelemek között. Ha azt akarjuk, hogy a felhasználó TAB karaktereket is beilleszthessen a Memo komponensbe, akkor állítsuk a WantTabs tulajdonságot True-ra. Ha a False alapértéket hagyjuk meg, akkor a CTRL-TAB kombináció helyettesíti a tabulátort. A WantReturns tulajdonság alapértéke True. Ez azt jelenti, hogy a felhasználó az ENTER segítségével különítheti el a sorokat. Ha ezt False-ra állítjuk, akkor az ENTER-t a form kapja meg feldolgozásra, és az ENTER szerpét a CTRL-ENTER kombináció veszi át.

A TButton és a TBitButton komponens

A nyomógombok használata meglehetősen elterjedt a Windows alkalmazások körében, rájuk kattintva valamilyan funkciót hívhatunk meg. (Természetesen más komponensek is képesek észlelni, ha az egérrel rájuk kattintunk, és így írhatunk hozzájuk kapcsolódó OnClick eseménykezelő eljárást.) A Delphi által támogatott két leggyakrabban használt nyomógombfajta a TButton és a TBitBtn.

A TButton komponens a komponenspaletta Standard oldalán található, míg a bittérképes nyomógomb-komponens (TBitBtn) az Additional oldalon. Mind az előbbi, mind az utóbbi komponens valóságos nyomó-

gomb illúzióját kelti. Ha a felhasználó rájuk kattint, vagy lenyomja a szóközt, amikor a fókusz rajtuk van, akkor benyomódnak. A TBitBtn abban különbözik az egyszerű nyomógombtól, hogy azon bittérképes grafika is megjeleníthető a szövegen kívül. Választhatunk a Delphi saját, előregyártott bittérképei (pl. a Delphiben is használt OK, Cancel vagy Help gombok) közül, és használhatunk saját készítésűeket is.

A sima és ábrával ellátott nyomógombok legáltalánosabb közös tulajdonságairól ejtünk először szót. Aztán röviden leírjuk a bittérképes gombok legfontosabb tulajdonságait.

Cancel. A Cancel tulajdonság True-ra állításával elérhetjük, hogy a nyomógombot az ESC billentyűvel is le lehessen nyomni. Ha értéke True, a Delphi minden alkalommal, amikor a felhasználó lenyomja az ESC billenyűt, lefuttatja a nyomógomb OnClick eseménykezelőjét. Célszerű ezt pl. egy olyan ablak Cancel gombjához kötni, amelyben a felhasználó megerősítheti állománytörlési szándékát.

Default. Ennek a tulajdonságnak a True-ra állításával rendelkezhetünk arról, hogy az ENTER billentyű lenyomására automatikusan végrehajtódjon a gomb OnClick eseménykezelő eljárása. A Default tulajdonsággal rendelkező nyomógombot alapértelmezés szerinti nyomógombnak hívják.

Megjegyzés: Ha a felhasználó lenyomja az ENTER-t, miközben az inputfókusz egy nyomógombon van, akkor a gombhoz tartozó OnClick esemény váltódik ki. Ennek bekövetkezése független attól, hogy van-e az aktuális formon alapértelmezés szerinti nyomógomb. Hasonlóan, ha a fókusz egy olyan TMemo komponensen van, amelynek WantRenturns tulajdonsága True értékű, akkor a kocsi vissza karaktert a TMemo komponens kapja meg. (Itt megint csak mellékes, hogy van-e alapértelmezés szerinti nyomógomb.)

TBitButton tulajdonságok

Az alábbiakban a bittérképes nyomógombok jellemző tulajdonságait ismertetjük.

Glyph, NumGlyphs. A Glyphs tulajonsággal jelölhetjük ki a grafikus nyomógombon megjelenítendő bittérképet. Ha fejlesztési időben állítjuk be, akkor a Delphi lehetővé teszi, hogy egy szokásos Windows Open dialógusablakból válasszuk ki az állományt, amely a bittérképet tartalmazza. Egy nyomógombon akár négy képet is használhatunk, de mindnek ugyanolyan méretűnek kell lenni. A NumGlyph tulajdonsággal ezek számát állíthatjuk be. (Részletesebben l. a beépített súgót.)

Kind. A Kind tulajdonság meghatározza a gombon megjelenő bittérképes grafika fajtáját. Például a bkOK érték beállítása esetén a Delphiben is használt, zöld pipát ábrázoló „OK" gomb lesz látható. Itt 11 különböző értéket állíthatunk be. Válasszuk a bkCustom értéket, ha saját bittérképet szeretnénk használni (és a Glyph tulajdonságot is megfelelően állítsuk be).

Layout, Margin, Spacing. A Layout tulajdonság határozza meg, hogy az ábra a gomb jobb, bal, felső vagy alsó oldalán helyezkedjen-e el. A Margin tulajdonság a gomb és a kerete közti hely nagyságáért felelős. A Spacing tulajdonság értéke pedig a felirat és a keret közti hely méretét határozza meg.

A TListBox és a TComboBox komponens

A TListBox (listaablak) és a TComboBox (csatolt listaablak) segítségével görgethető listát jeleníthetünk meg, a lista elemei közül a felhasználó kiválaszthatja azt, amelyikre éppen szüksége van. A különbséget az jelenti, hogy a listaablakban nem változtathatjuk meg az elemeket. (A csatolt listaablak biztosít egy szerkesztési területet, amelybe a felhasználó közvetlenül írhat adatokat.) Az elemek megadása az Items tulajdonság beállításán keresztül történik, amit kétféleképpen tehetünk meg: az objektumfelügyelőben és programon keresztül. Ehhez a két komponenshez még visszatérünk a 6. és 13. fejezetben, ahol azt is megtudhatjuk, hogyan lehet képeket helyezni a listaablakra.

A listaablakokhoz leggyakrabban az OnClick esemény kapcsolódik. Ezt nem csak a listaelemekre történő kattintás váltja ki, hanem az is, ha a felhasználó a billentyűzet segítségével mozog a lista sorai között. Emiatt

gyakran arra kényszerülünk, hogy az aktivizáló programkódot inkább az OnDblClick eseményhez kössük, mintsem az OnClick-hez. (Egyébként a Windows szabvány is ez, kivételt képeznek ez alól a Windows 95 Start menüjéhez hasonló elemek.)

Most ismertetjük ezen komponensek legfontosabb tulajdonságait, amelyek beállítására szükségünk lehet fejlesztés közben.

Tipp: Az IntegralHeight tulajdonság True-ra állításával elérhetjük, hogy ne lehessen a listaablakot úgy átméretezni, hogy egyes elemek csak részben látszódjanak.

Columns. A Columns tulajdonság csak a sima listaablakokra vonatkozik, ez határozza meg a felhasználó által látható oszlopok számát az ablakban.

ExtendedSelect, MultiSelect. Ezek a tulajdonságok szintén csak a sima listaablakok sajátjai. Ha a MultiSelect True értékre van állítva, akkor a felhasználónak lehetősége van egynél több elemet kiválasztani a listáról a megszokott Windows technikákat használva. Ha mind az ExtendedSelect mind a MultiSelect értéke True, akkor a felhasználó a szomszédos elemek egy csoportját a SHIFT nyomva tartása mellett, míg szomszédos elememek különálló csoportjait a CTRL billentyű nyomva tartása mellett választhatjuk ki.

Sorted. Mindkét komponens rendelkezik ezzel a tulajdonsággal. Ennek segítségével a lista rendezettségét biztosíthatjuk, új elem bekerülése esetén a sima lista vagy a csatolt lista automatikusan átrendeződik.

Style. Ez a tulajdonság felelős a csatolt lista megjelenési formájáért. Öt érték közül választhatunk. Ezek közül három igen gyakran használt. Az első – egyben az alapérték is – a csDropDown. Ez jelenti a jól megszokott szerkesztési területtel ellátott gördülőlistát. Ha a csDropDownList értéket állítjuk be, akkor ugyanezt kapjuk szerkesztési terület nélkül. A csSimple beállítással nem kapunk adatokat tartalmazó listát.

Megjegyzés: Ha az előbbi tulajdonságnak lbOwnerDrawFixed vagy lbOwnerDrawVariable értéket adunk, akkor az ablakban lehetőség nyílik szöveg mellett grafika használatára is.

A TCheckBox és a TRadioButton komponens

Ezeket a komponenseket akkor célszerű használni, ha a felhasználó számára nem kizárólagos választási lehetőséget szeretnénk biztosítani. A programkódba lehet beépíteni (a Checked tulajdonság beállításával), hogy kezdetben az ablakban az adott opció mellett kijelölt (beikszelt) vagy üres négyzet szerepeljen-e. Másrészt ha egymást kölcsönösen kizáró lehetőségeket szeretnénk a felhasználó elé tálalni, érdemes rádiógombokat használnunk. (Erre példa a Delphi Align dialógusablaka.) Ha a felhasználó rákattint egy csoport egy rádiógombjára, akkor a többi rádiógomb automatikusan kikapcsolt állapotba kerül. Amikor a felhasználó egy kijelölőnégyzetre vagy egy rádiógombra kattint, akkor – az elvárásoknak megfelelően – a komponens OnClick eseménye végrehajtódik.

Megjegyzés: Mivel a rádiógombok csoportonként csatoltak, egy form két rádiógombja egyetlen esetben lehet egyszerre bekapcsolt állapotban: amikor azok különböző tartalmazó komponensekben vannak.

A TTimer komponens

A TTimer időzítő komponenst akkor használhatjuk, ha szeretnénk, hogy valami (vagy semmi, ha csak szünetet szeretnénk) periodikusan végrehajtódjon. Ezt a komponenst használnánk például abban a programban, amelyik minden két órában elindul, és ellenőrzi a részvényárfolyamokat. Egy hétköznapibb példánál maradva: ha egy óra megjelenítéséhez az alkalmazásunkban, szükség van a megjelenített óra percenkénti vagy másodpercenkénti felfrissítésére. (Az ehhez szükséges egyetlen programsort l. a fejezet „Az OnTime esemény" részében.)

Az időzítő komponens láthatatlan a felhasználó számára, az ehhez tartozó ikon csak fejlesztési időben jelenik meg. Ezért nincs jelentősége, hová helyezzük az időzítőelemet fejlesztés közben. Az időzítő hasznos segédeszköz a Delphi programozó számára, használatát azonban nem célszerű túlzásba vinni. A Windows 3.X maga is korlátozza az egyidőben futó alkalmazásokban (nem csak a Delphi alkalmazásokban) használható időzítők számát 16-ra. A komponens túlzott használata értékes Windows erőforrások lekötését eredményezi.

Az időzítő ikonja egy régimódi órát ábrázol, amely a komponenspaletta System oldalán található meg. Két fontos tulajdonság kapcsolódik ehhez a komponenshez: az Enabled és az Interval.

Enabled

Az enabled logikai értékeket vehet fel (True/False). Ez a tulajdonság határozza meg, hogy az óra elindulhat-e. Ha értékét True-ra állítjuk fejlesztés alatt, akkor a form betöltésekor a stopper rögtön elkezd ketyegni. (Természetesen a „ketyegés" itt metaforikusan értendő, nem kíséri tényleges hang a működését, hacsak nem írunk erre külön programot.) Mivel az időzítőkomponensek láthatatlanok a felhasználó számára, ezért könnyen elképzelhető, hogy az nem is tud az időmérésről. Ezért gyakran célszerű értesíteni a felhasználót a stopper működésbe lépéséről egy üzenetablakon, egy grafikus vezérlőn vagy egy óra ábráját tartalmazó képablakon keresztül.

Ha az Enabled tulajdonságot False-ra állítjuk a tervezési idő alatt, akkor a stoppert a programban úgy indíthatjuk el, hogy a tulajdonságnak True értéket adunk. Hasonlóan, ha a programban False-ra állítjuk az Enabled értékét, az időmérés véget ér. (Célszerű az időméréshez kapcsolódó eseménykezelő eljárásban a stoppert ideiglenesen – az eljárás futási idejére – megállítani.)

Interval

Az Interval tulajdonság határozza meg, hogy a Delphi mennyi ideig várjon a Timer eseménykezelő eljárás (l. következő fejezetrész) hívásáig. Ezt az időközt milliszekundumban (egy másodperc ezredrésze) kell megadni, az eleméleti határokat az 1 és a 65 535 milliszekundum (egy kicsivel több, mint 1 perc 5 másodperc) jelenti. Annak, hogy ezek csak elméleti határok az az oka, hogy a hardver másodpercenként csak 18 jelet produkál. Így ezek a jelek kevesebb, mint 56 milliszekundumonként követik egymást, ezért ennél kisebb időközök használatának nincs igazán értelme. Két olyan időközbeállítás, amelyek nem különböznek legalább 56-tal, ugyanahhoz az eredményhez vezetnek. (Kisebb intervallumok használatát az API függvények teszik lehetővé. L. 6. fejezet.)

Minél rövidebb időközt adunk meg, annál több processzoridő fordítódik az időmérő eseménykezelőjének aktivizálására. Ha túl kicsi időközt állítunk be, gépünk csigalassúvá válhat.

Megjegyzés: Ha az Interval tulajdonságot 0-ra állítjuk, az időmérés abbamarad.

Az OnTimer esemény. A Delphi az Interval tulajdonságban beállított értéknek megfelelő gyakorisággal hajtja végre az OnTimer eseménykezelő eljárást. Elképzelhető azonban, hogy a CPU éppen foglalt, amikor a megadott intervallum által meghatározott idő eltelik, így nem garantálható, hogy a Delphi pont a kívánt időben hívja meg az OnTime eseménykezelőt. (A Delphi tudja, hogy mikor telik le az idő, csak lehet, hogy még be kell fejeznie a „dolgát" mielőtt aktivizálhatná az eljárást.) Ha a beállított idő eltelt és a processzor szabad, a Delphi azonnal meghívja az eljárást. (Lehetőség van olyan programrész írására, amelynek segítségével megállapíthatjuk, hogy a tervezettnél több idő telt-e el.)

A következő példaprogram egy olyan órát valósít meg, amely minden másodpercben felfrissíti magát. A form létrhozásához kövessük az alábbi lépéseket:

1. Az üres formhoz illesszünk fejlécet és adjuk hozzá az időmérő komponenst.
2. Állítsuk a címsor AutoSize tulajdonságát True-ra és a font méretet pedig elegendően nagyra. Az időzítő komponens Interval tulajdonságának értékét állítsuk 1000-re (1000 milliszekundum =1 másodperc).

Ezután a következő kódot írjuk a Timer Timer1 nevű eseménykezelőjébe:

```
Procedure TForm1.Timer1Timer(Sender: TObject);
begin
Label1.Caption = 'Az idő: ' + DateTimeToString(Now);
end;
```

A Delphi durván minden másodpercben meghívja ezt az eljárást, és felfrissíti az órát, mivel az Interval tulajdonságnak 1000-et adtunk értékül. (Részletesebben a 6. fejezetben ismerkedhetünk meg az itt használt dátum/idő függvényekkel.)

Tipp: *Ha azt szeretnénk, hogy egy OnTimer eseménykezelő eljárás ritkábban csináljon valamit, mint ahogy azt a leghosszabb beállítható időköz engedné (percenként egyszer), akkor egy statikus változó (típusos konstans) bevezetésével érhetünk célt. A változóban tárolhatjuk az eltelt időintervallumok számát. Az 5. fejezetben láthatunk több példát az ilyen típusú változókra.*

A TStringGrid és a TDrawGrid komponens

A TString komponens segítségével szöveges táblázatot hozhatunk létre. A TDrawGrid táblázatkomponensben grafikát és szöveget egyaránt megjeleníthetünk. A sorok és oszlopok találkozását általában *celláknak* hívják. A két táblázatkomponens tulajdonságainak többsége hasonlóan működik. Most csak a leglényegesebb tulajdonságokra térünk ki. Sok tulajdonság valódi értelme csak futási idő alatt mutatkozik meg. Például a Cell tulajdonság kettős funkcióval bír: lehetővé teszi a cellákba történő írást, és hozzáférést biztosít a cellákban található adatokhoz. A táblázatokhoz kapcsolódó legfontosabb esemény az OnDrawCell. Az eseményt az váltja ki, ha a felhasználó kijelöli valamelyik cellát. (A táblázatokról bővebb felvilágosítást a beépített help nyújt.)

ColCount, RowCount. Ezek a tulajdonságok határozzák meg a táblázatban szereplő oszlopok és sorok számát. Mindkét tulajdonság alapértéke 5. Az értékek átállítását az objektumfelügyelőn keresztül ill. programban tehetjük meg.

DefaultColWidth, DefaultRowHeight, Height, Width. A DefaultColWidth és a DefaultRowHeight tulajdonságok felelősek az oszlopok szélességéért ill. a sorok magasságáért. Height és a Width tulajdonságok a táblázat egészének a magasságáért és a szélességéért felelősek.

FixedCols, FixedRows, FixedColor. Táblázatok használatakor gyakran előfordul, hogy azt szeretnénk, ha bizonyos cellákat állandó információk megjelenítésére foglalhatnánk le. Például szükség lehet arra, hogy az oszlopok fejlécei állandóan láthatóak maradjanak függetlenül attól, hogy a felhasználó a munkalap melyik részével dolgozik. A rögzített oszlopok és sorok mindig szürkék, és a táblázat bal, illetve felső szélén találhatók. A

FixedColor tulajdonság határozza meg azt a színt, amelyet a rögzített cellák kapnak.

GridLineWidth. Ez a tulajdonság határozza meg a cellákat elválasztó vonalak vastagságát. Nagyobb értékek vastagabb vonalat jelentenek, a nulla értékkel pedig törlődnek a cellák közti vonalak. (Így is láthatók maradnak a cellahatárok, azokat a cellák körüli fehér keret jelzi majd.)

Options. Ez a tulajdonságcsoport 15 logikai (True/False) értéket takar. Ezek főleg a felhasználó elé kerülő táblázat megjelenési formájáért és annak futási időben történő alakíthatóságáért felelősek. Ha például a goEditing altulajdonságot True-ra állítjuk, akkor a felhasználó megváltoztathatja a cellák tartalmát. (Az Options tulajdonságról bővebb információkat a beépített súgóban találhatunk.)

■ *Menük tervezése*

A menüelemek nem mások, mint speciális komponensek, amelyeket formjainkra illeszthetünk. A menüpontok csak a kattintásra reagálnak. A menüpontok alkalmas megválasztása nélkülözhetetlen a felhasználóbarát alkalmazások készítéséhez. A Delphiben lényegében akármilyen szintű menürendszert építhetünk, valamint lehetőség van felbukkanó-menük készítésére is.

Az almenüket is tartalmazó menürendszert hierarchikus menüknek nevezik. Túl sok menüszint használata áttekinthetetlenné teheti programunkat a felhasználó számára. Négy szint már biztosan sok, két-három szint az, amivel a legtöbbször találkozhatunk. A felhasználónak a menüpont mellett látható jobbra mutató háromszög mutatja, hogy abból a menüpontból egy almenü nyílik.

Tipp: *A sok almenü helyett gyakran érdemesebb a célnak megfelelő dialógusablakokat használni.*

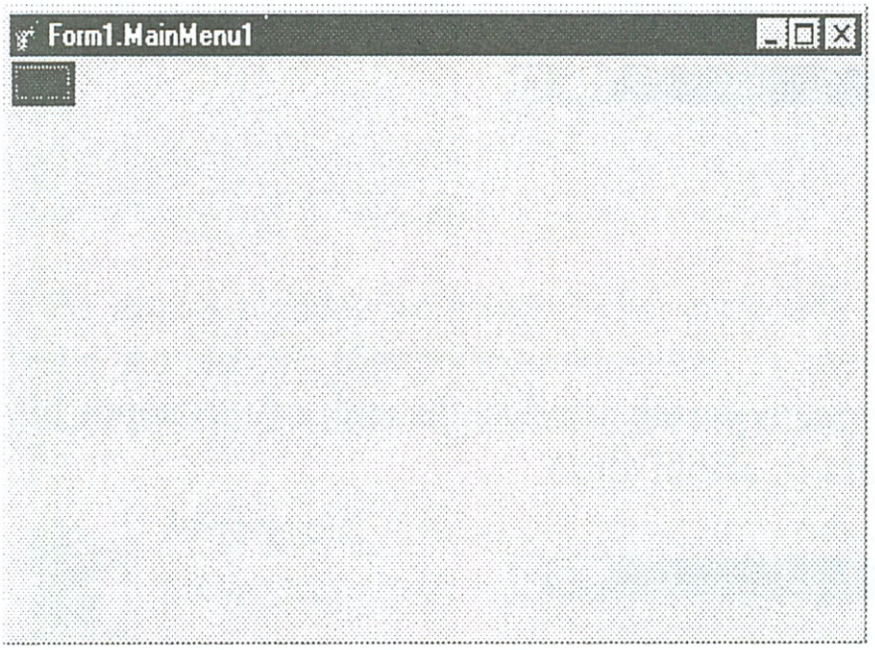
Ahhoz hogy projektünk menüvel rendelkezzen, első lépésként adjunk egy menükomponenst (vagy felbukkanó menükomponenst) a formhoz. A

TPopupMenu komponens segítségével egy olyan menüt hozhatunk létre, amelyet a felhasználó az egér jobb oldali gombjával aktivizálhat, amikor a kurzor a form egy adott komponensén vagy magán a formon van. A komponensekhez PopupMenu tulajdonságukon keresztül rendelhetünk különböző felbukkanómenüket. A kívánt menükomponenseket (TMenu vagy TPopupMenu) úgy illeszthetjük a formhoz, ha azokra egyszerűen kettőt kattintunk az egérrel. Ebben az esetben nem számít, hogy hol jelenik meg az adott menükomponens a formon.

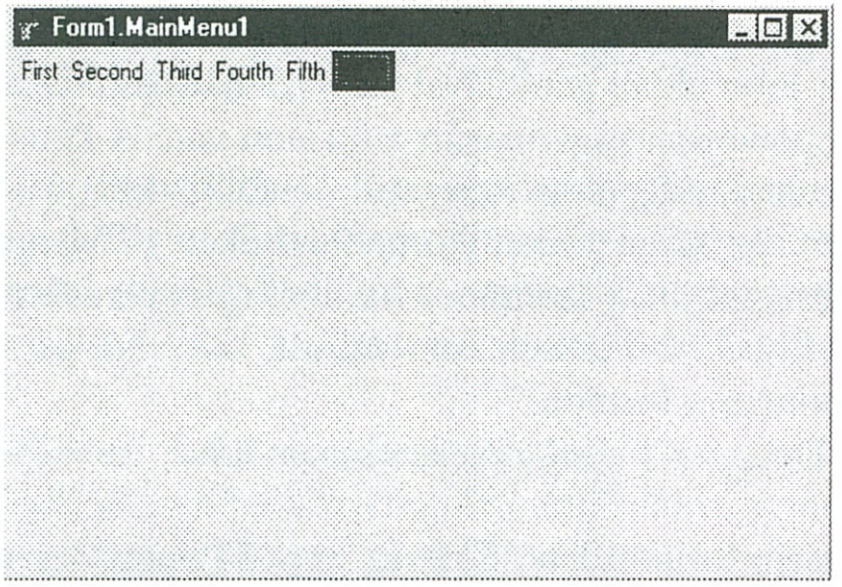
Következő lépés a Menu Designer (menütervező) megnyitása. Ezt kétféleképpen is megtehetjük:

- Kattintsunk kettőt a már a formon lévő TMenu vagy TPopupMenu komponensre.
- Kattintsunk kettőt a menükomponens Items tulajdonságára (vagy kattintsunk a felsorolásjelre).

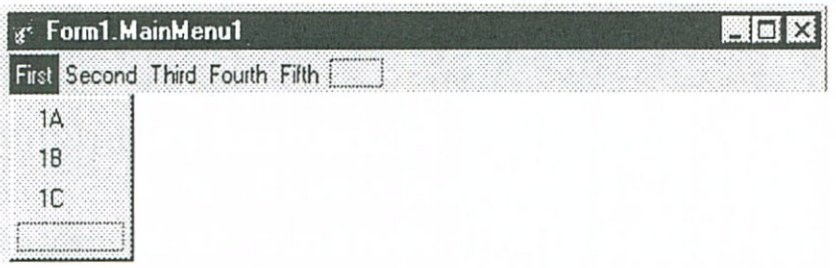
A képernyőn megjelenik a menütervező (l. alábbi ábra).



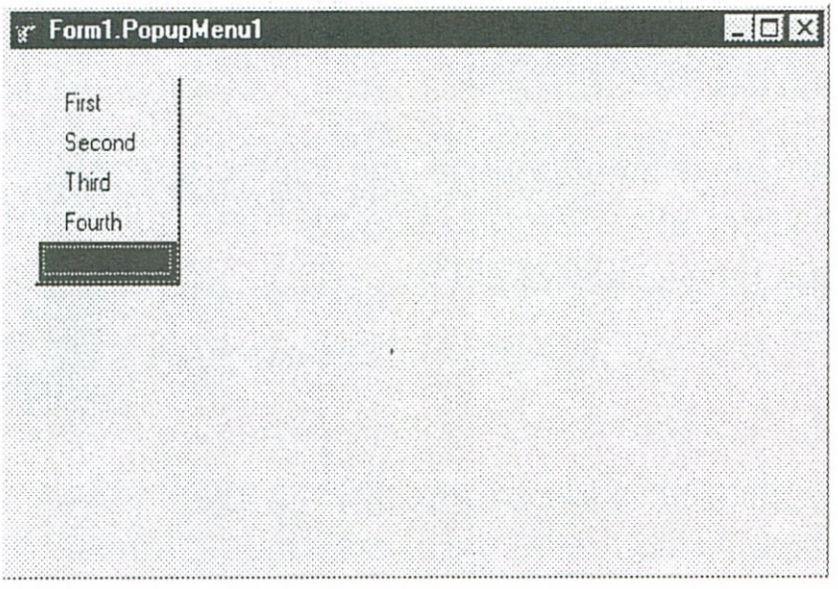
Vegyük észre, hogy az első (üres) menüpont ki van jelölve a menütervezőben. A menütervezőben képet alkothatunk menünk futási idő alatti megjelenési módjáról. Figyeljük meg az alábbi ábrát.



Látható, hogy a menüpontok úgy jelennek itt meg, ahogy azt majd a formon is teszik. Természetesen a főmenüpontok alá is készíthetünk további menüket. Az alábbi ábrán a főmenü első menüpontjából nyíló almenüre láthatunk példát.

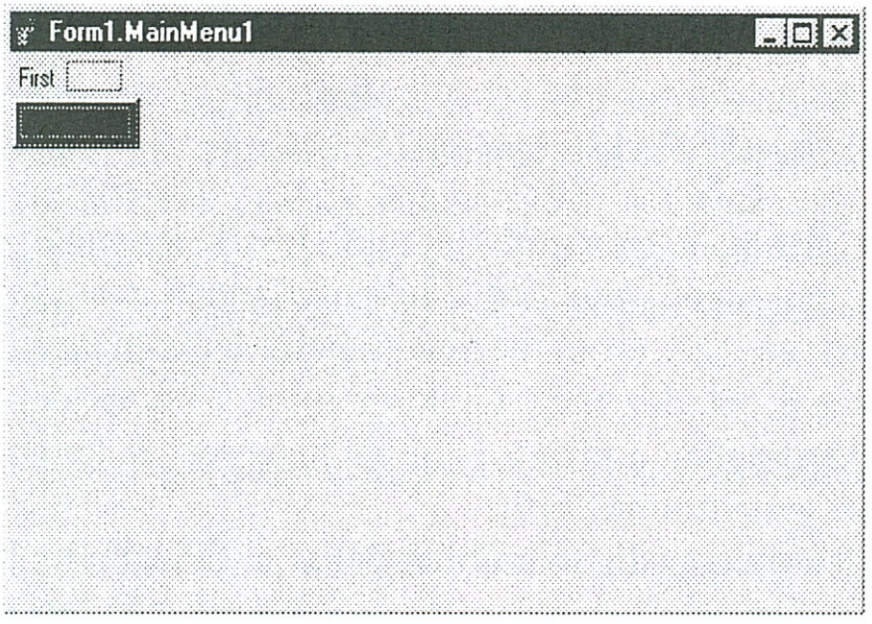


Felbukkanó-menükomponens esetében a Delphi csak egy oszlopot biztosít a menüpontok számára (l. alábbi ábra).



A menütervező használata

Tegyük fel, hogy egy főmenű-komponenssel dolgozunk. Az ide beírt szöveg lesz a menüponthoz tartozó felirat – és természetesen a Caption tulajdonság értéke is. Az ENTER billentyű lenyomásával pontozott keret jelenik meg a menüpont alatt és annak jobb oldalán, ahogy azt az alábbi ábra is mutatja.



Az alulsó téglalap az almenü első elemének felel meg, a jobb oldaliba pedig a második főmenüpont kerülhet. A menütervezőben egér vagy a nyilak segítségével mozoghatunk a menüpontok között.

Megjegyzés: *A menüpont Name (név) tulajdonsága a Caption (felirat) tulajdonság értékét kapja meg. A programkódot áttekinthetőbbé tehetjük, ha a Name tulajdonságot átírjuk az objektum felügyelőben.*

Egy adott menüpont beillesztése után lehetőség nyílik tulajdonságainak megváltoztatására. A fent említet Caption és Name tulajdonságok kivételével a leglényegesebbeket rövidesen ismertetjük. A menüpont mellett megjelenő felsorolásjelet szintén a Caption tulajdonságban állíthatjuk be. A három pont jelzi, hogy a menüpontra kattintva egy dialógusablak jelenik meg. Gyakran célszerű tagoló vonalakat használni egy menüponthoz tartozó almenüben, ennek egyik nagy előnye az, hogy segítségével a hosszú almenüket csoportosítani lehet. Ha öt-hat pontnál több található egy

menüben, akkor érdemes megvizsgálni, hogy van-e lehetőség ezek valamilyen csoportosítására.

Tipp: *Ha egy menü Caption tulajdonságának elválasztójelet (-) adunk értékül, akkor az egy tagolóvonal megjelenését eredményezi.*

Megjegyzés: *Mindkét típusú menü kizárólag az OnClick eseményre reagál.*

A gyorsító- és parancsbillentyűk kijelölése

Ahogy azt korábban már láttuk, a gyorsítóbillentyűk teszik lehetővé a felhasználó számára, hogy egy menüparancsot a billentyűzetről is aktivizálni lehessen, nevezetesen az ALT és a megfelelő betű lenyomásával, amelyet a programkódban a & jel előz meg. Csakúgy mint a komponensek esetén, a Caption tulajdonságban a kívánt betű elé helyezett & karakterrel jelölhetjük ki az adott gyorsítóbillentyűt.

A parancsbillentyűk segítségével pedig menüfunkciókat aktivizálhatunk a menü megnyitása nélkül. Egy adott menüponthoz a következőképpen jelölhetünk ki parancsbillentyűt:

1. Válasszuk ki a menüpontot.
2. Adjuk a kívánt billentyűkombinációt értékül a menüpont ShortCut tulajdonságának. (Az objektumfelügyelő jobb oldali oszlopából nyíló listaablakban láthatjuk a leggyakoribb parancsbillenyűket.)

Figyelem! *A Delphi nem ellenőrzi, hogy egy adott gyorsító- vagy parancsbillentyű már korábban előfordult-e.*

Tipp: *Az ALT-F4 nincs a parancsbillentyűk listáján, mert ennek funkciója minden formra ugyanaz (bezárja azt), hacsak nem távolítottuk el a rendszermenüt. Ha mindenképp jelezni akarjuk azt, hogy az Exit elem parancsbillentyűje az ALT-F4, akkor adjuk azt közvetlenül a Caption tulajdonsághoz.*

A Checked tulajdonság

Ezzel a kétértékű (True/False) tulajdonsággal állíthatjuk be, hogy egy kis pipa megjelenjen-e az adott menüpont előtt. Értékét általában futási időben változtatjuk meg a felhasználó által végzett műveletek tükrében. Fejlesztési időben az alapbeállítást definiálhatjuk.

A Break tulajdonság

Segítségével többoszlopos menüket készíthetünk. Alapértéke mbNone, mbBreak re állításakor a Delphi egy új oszlopot kezd, amelynek első eleme az aktuális menüpont lesz. Ha a Break tulajdonságnak az mbBarBreak értéket adjuk, akkor a Delphi egy függőleges elválasztó vonalat is helyez az oszlopok közé (mint a Windows 95 Program pontjában). Nyilvánvaló, hogy az alapérték megváltoztatásának csak almenük esetében van értelme.

Az AutoPopup tulajdonság (csak felbukkanó-menük esetén)

Ha értéke True, az egér jobb oldali gombjának lenyomására felbukkan a menü. Alapértéke True.

Menüpontok beszúrása és törlése

A Delphi menütervezőjében lehetőség van menüpontok beszúrására és törlésére. Nézzük meg először, hogyan lehet beilleszteni egy új menüpontot egy meglévő menübe. Az új menüpont, a főmenü esetén a kijelölt menüelem bal oldalára, míg ha egy almenüben vagyunk a kijelölt menüelem fölé kerül.

Új menüpont beillesztésének módja a következő:

- Állítsuk a kurzort a kiválasztott menüpontra, és nyomjuk le az ENTER-t

Menüpontot a következőképpen törölhetünk a menüből:

- Jelöljük ki a menüpontot, és nyomjuk le a DEL billentyűt.

Menüpontok átrendezése

Fejlesztési idő alatt a menüelemeket a Windows „fogd és vidd" módszerével egyszerűen mozgathatjuk. Ez alól van egy természetesnek tűnő kivétel: nem mozgathatunk át egy menüpontot az almenüjébe. A menüpontok mozgatása közben a kurzor alakja megváltozik.

Megjegyzés: Egy menüpontot csak az ahhoz tartozó almenükkel együtt lehet áthelyezni.

Például egy főmenüpont áthelyezését a főmenü egy másik helyére úgy valósíthatjuk meg, hogy azt az egérrel addig húzzuk (a bal oldali gomb nyomva tartásával), amíg a kurzor hegye a kívánt helyen nem áll, ekkor engedjük el az egérgombot. Ha a menüpontot egy almenübe szeretnénk áthelyezni, akkor azt kell kihasználnunk, hogy mozgatás során a Delphi kinyitja azt a menüt, ahová a kurzor hegye mutat.

Beágyazott menük készítése

A Delphiben lehetőség van olyan menük készítésére, amelyben menüpontok újabb menüt takarnak. Ezt a menüpont jobb oldalán látható nyíl mutatja. Az ilyen beágyazott menük létrehozásához először is ki kell jelölnünk, hogy melyik menüpontból nyíljon az almenü. Ezután az alábbi lépések követésével érhetünk célt:

1. A menütervezőben nyomjuk le a jobb oldali egérgombot, és válasszuk a Create Submenu pontot a gyorsmenüből.
2. Adjuk meg az első almenüpont nevét.
3. Nyomjuk le az ENTER-t (vagy a LEFELÉ-NYILAT, amikor a fókusz a menüszerkesztőn van), hogy helyet csináljunk a következő almenüpontnak.

Ezt addig folytassuk, amíg elő nem állítottuk a kívánt méretű almenüt. Ha kész, nyomjuk le az ESC billentyűt, hogy visszatérjünk az előző szintre. (Természetesen itt is áthelyezhetjük a már meglévő menüpontokat.)

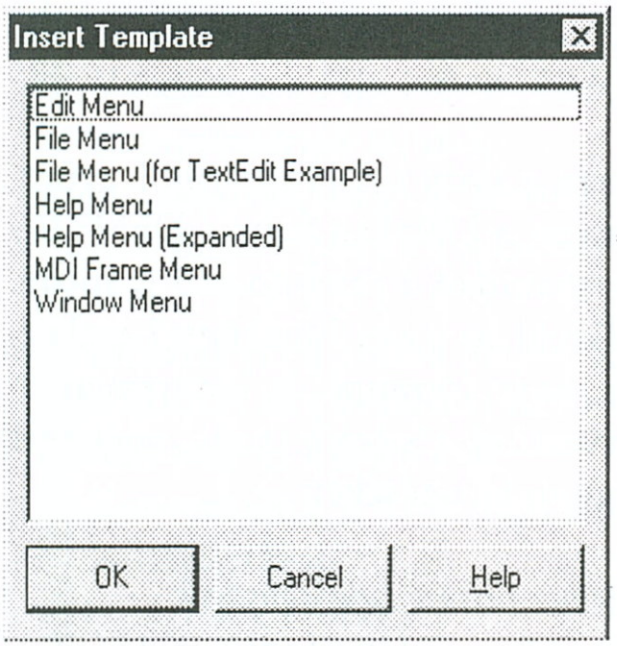
Tipp: Ha a menütervezőben egy meglévő menüstruktúrát áthelyezünk, akkor – mivel az almenük is vele mozognak – automatikusan egy beágyazott almenühöz jutunk.

Menüsablonok

A Delphi tartalmaz néhány előre elkészített menüsablont; a felhasználó is készíthet újra hasznosítható menümintákat. Amennyiben a sablonok közül szeretnénk menüt választani, a következőket kell tennünk:

1. Nyomjuk le a jobb oldali egérgombot a gyorsmenü megnyitásához.
2. Válasszuk a gyorsmenü Insert From Template pontját.

Ekkor kinyílik az Insert Template dialógusablak, amely az éppen elérhető menüsablonok listáját tartalmazza (l. alábbi ábra).



Válasszuk ki a kívánt menümintát. (Az eddig tanultak alapján ezután már nem okozhat gondot a menü igény szerinti elrendezése.)

Megjegyzés: Szükség van a menüsablon elemeihez tartozó OnClick eseménykezelők átírására, és ha a felhasználónak nem tetszenek a nevek, amelyeket a Delphi a menüelemeknek adott, akkor a Name tulajdonságok beállításával megváltoztathatjuk azokat.

Ha lehetővé kívánjuk tenni, hogy egy általunk elkészített menü később is felhasználható legyen, akkor kövessük az alábbi pontokat:

1. Nyomjuk le a jobb oldali egérgombot a gyorsmenü megnyitásához.
2. Válasszuk a Save As Template pontot.

Megjelenik a Save Template dialógusablak, ahol a menü rövid leírását adhatjuk meg. Itt nem kell követnünk a megszokott DOS-os/Windows-os elnevezési konvenciókat.

Menüpontok érzékennyé tétele

Menüpontok a külső események közül csak a kattintásra reagálnak. Az eseménykezelő eljárás sablonját a következőképpen kapjuk meg.

- Kattintsunk kettőt a kívánt menüpontra a menütervezőben.

Vagy amennyiben a formon belül vagyunk:

- Kattintsunk a menüpontra.

■ *Formok ASCII reprezentációja*

Egy form ASCII reprezentációja tartalmazza a form tulajdonságaink szöveges leírását. Hasznossága vitathatatlan, hiszen a Delphi a form (.DFM) fájlokat bináris formában tárolja, amely olvashatatlan a programozó számára. A form szöveges ábrázolása egyben rendkívül hasznos hibakereső eszköz. Használatával könnyedén ellenőrizhetjük, hogy formunk és komponenseinek tulajdonságai fedik-e azt, amit elterveztünk. Szintén nagy segítséget jelent egy komplex projekt verzióinak nyomon követésében.

Tipp: *Egy formhoz vagy komponenseihez tartozó tulajdonságok módosítását egy egyszerű szövegszerkesztőben is elvégezhetjük az ASCII reprezentáció használatával. (Fontos azonban tudnunk, hogy mielőtt a szövegszerkesztőbe beolvasnánk a fájlt, konvertálnunk kell szöveges állománnyá. A Delphi képes a form jellemzőket .DFM formátumban fogadni, szöveges fájlként elmenteni és fordítva. Mindezt figyelembe véve mégiscsak jobb ha a Delphi beépített szövegszerkesztőjénél maradunk.)*

A konvertáláshoz először meg kell nyitnunk a formot. Fontos viszont, hogy egy formot nem nyithatunk meg egyszerre .DFM (vizuális) és .TXT (szöveges) formátumban is. Ez azt jelenti, hogy leggyakrabban egy már

elmentett .DFM állományt kell szöveggé konvertálnunk. Az alábbi pontok írják le ezt a műveletet.

1. Válasszuk a File | Open File menüpontot.
2. Jelöljük ki a szöveggé alakítandó formot a megjelenő dialógusablakban. A Delphi beolvassa az állományt a fejlesztői környezetbe ASCII formátumban.
3. Ekkor válasszuk a File | Save File As pontot és adjunk meg .TXT kiterjesztést az elmentendő fájlnak.

Ha egy form fájl szöveges változatát bináris (.DFM) formátumúvá szeretnénk alakítani a következőket kell tennünk:

1. Válasszuk a File | Open File menüpontot.
2. Jelöljük ki a szöveges állományt, amelyet bináris formátumúvá szeretnénk alakítani a megjelenő dialógusablakban. A Delphi beolvassa az állományt a fejlesztői környezet kódszerkesztőjébe ASCII formátumban.
3. Válasszuk a File | Save File As pontot és adjunk .DFM kiterjesztést az elmentendő fájlnak.

Nézzünk most meg egy form ASCII reprezentációját!

1. Kezdjünk egy új projektet és a formunk fejlécébe írjuk az ASCII nevet.
2. Illesszünk egy sima, alapméretű, alaphelyzetű TButton komponenst, a Button alapbeállított név használata mellett.
3. Mentsük el a form fájlt.
4. Konvertáljuk a form fájlt szövegessé az előbb bemutatott technika használatával.

A fájl így fog kinézni:

```
object Form1: TForm1
  Left = 200
  Top = 99
  Width = 435
  Height = 300
  Caption = 'ASCII'
  PixelsPerInch = 96
  object Button1: TButton
    Left = 169
    Top = 120
```

```
    Width = 89
    Height = 33
    Caption = 'Button1'
    TabOrder = 0
  end
end
```

Látható, hogy az ASCII reprezentáció a form és a TButton tulajdonságok szöveges leírását tartalmazza. Például az alábbi kódsor a form új fejlécéért felelős:

```
Caption = 'ASCII'
```

A szöveges állomány komponenst leíró része így indul:

```
object Button1: TButton
```

Ez jelzi, hogy melyik *osztályba* tartozik az objektum. A 7. fejezetben erről többet tudhatunk majd meg.

5. FEJEZET

A Delphi programozás alapjai

Az előző fejezetekben eljutottunk addig, hogy tudjuk, hogyan néz ki egy Delphi alkalmazás. Láttuk, hogyan építhetjük fel formjainkat komponensek hozzáadásával, és belekóstolhattunk a Delphi alkalmazások gerincét képező eseménykezelő eljárások írásába is. Lényeges azonban hangsúlyozni, hogy az eddig megismert eseménykezelők igen egyszerűek voltak. Ahhoz, hogy ezeknél bonyolultabbakat is meg tudjunk írni, meg kell ismerkednünk a Delphiben használt Object Pascallal. Ez a fejezet bevezetőül szolgál a nyelv elsajátításához. Az alapelemek – úgymint konstansok, változók, ciklusok és a Delphibe beépített döntési struktúrák – megismerése mellett megtanulhatjuk, hogyan kezelhetjük futási idő alatt projektjeinkben a Delphi objektumait a megfelelő programkód segítségével.

A tanulást a kódszerkesztő ablak használatával és a Delphi utasítások alaptípusaival kezdjük, majd áttérünk a tulajdonságok programon keresztül történő beállításának és az olyan alapfogalmak, mint változók, számok, konstansok tárgyalására. A következő lépcsőt az olyan alapvető programozási elemek megismerése jelenti, mint a típus, ciklus, és az if-then utasításpár. Ezt követi a Delphi objektumok kezelése futási időben. Ebben a részben például megismerhetjük a metódusokat. A metódusok gyakorlatilag olyan eljárások és függvények, amelyek a Delphi objektumok viselkedését befolyásolják futási idő alatt. (Ez a definíció nem teljesen precíz, mert vannak metódusok, amelyek úgy viselkednek, mint a tulajdonságok, és vannak tulajdonságok, amelyek úgy működnek, mint a metódusok!) Lesz szó olyan objektumokról – úgymint Screen, Printer, Application –, amelyek kizárólag futási időben érhetők el. Nem feledkezünk meg a Delphiben elérhető különböző üzenetablakokról (messagebox) és az egyszerű adatbekérést lehetővé tevő inputablakokról (inputbox) sem. A fejezet végén megmutatjuk azo-

kat a technikákat, amelyek segítségével egy alkalmazáson belül több formot is létrehozhatunk.

Mielőtt belevágnánk a fent vázolt témák ismertetésébe, fontos újra hangsúlyozni a Delphi programozás alaptételét: *a Delphi az egyes programrészleteket válaszként a különböző eseményekre hajta végre*. Fogjuk úgy fel, hogy a Delphi programok tulajdonképpen független építőkövekből állnak, amelyek akkor „kelnek életre", amikor a megfelelő esemény – amelyre érzékennyé tette őket a programozó – bekövetkezik. Lényeges itt még óva inteni azokat a programozókat, akik jártasak egy konvencionálisabb programnyelvben is, hogy annak metodikáját ne próbálják beleerőltetni a Delphi programok írásába. Ha más programozási nyelvekből hozott szokásainkat a Delphi programokra kényszerítenénk, akkor valószínűleg sok gonddal kellene szembenéznünk. Többek között nem jó, ha egy Delphi programra úgy gondolunk, mint aminek van egy első és egy utolsó sora, és a végrehajása fölülről lefelé történik. Ha mégis így teszünk, nem leszünk képesek kiaknázni a Delphiben rejlő lehetőségeket. A Delphi egy eseményvezérelt programozási környezet.

A programkódot mindig a kódszerkesztő ablakban írjuk. Ahogy már láttuk, ezt az ablakot úgy nyithatjuk meg, ha valamelyik komponensre, vagy a formra kettőt, vagy a Unit fülre egyet kattintunk (amely a Delphi 16-ban a form alatt, a 32-bites változatban pedig a form felett található). Egy másik lehetőség, ha az objektumfelügyelő egy eseményének sorában a jobb oldali cellára kattintunk. A kódablak részeit egy fejléc, amely a unit nevét tartalmazza, egy vagy több fül (a különböző oldalak számára) és a szerkesztési terület képezi. Az állapotjelző sor többek között megmutatja, hogy beszúró (INS) vagy felülíró (OVR) gépelési módban vagyunk-e, valamint a kurzor az ablak melyik sorában található.

Attól függően, hogy milyen opciót választunk, használhatunk különböző szerkesztő fajtákat: a szokásos Windows szerkesztőt, a hagyományos Turbo Pascal editort, a Brief valamint az Epsilon stílusú szerkesztőt. Az Options dialógusablakban (a Delphi 32-ben a Tools menüből a Delphi 16-ban pedig az Options | Environment menüből érhető el) található Editor oldalon beállíthatjuk – elképzeléseinknek megfelelően – a kódszerkesztő ablak különböző jellemzőit. (A Delphi 16 help-je ezt Editor options néven tartalmazza.) A kódszerkesztő ablakban a jobb oldali egérgomb lenyomása a gyorsmenü megnyitását eredményezi, amelynek segítségével gyorsan aktivizálhatunk egyes szerkesztési funkciókat.

Tipp: Egy adott, már megírt eseménykezelő eljárás kódjához gyorsan hozzáférhetünk, ha az objektumfelügyelő Events oldalán a megfelelő sor jobb oldali cellájába kettőt kattintunk.

■ *A Delphi parancsai*

A Delphi parancsok, mint a legtöbb programozási nyelvben, teljes „mondatoknak" tekinthetők. Nagy általánosságban mondható, hogy a Delphi parancsok *a Delphi objektumokkal csinálnak valamit*. Ezek az objektumok lehetnek komponensek tulajdonságai, numerikus kifejezések vagy még ezeknél is egzotikusabb struktúrák. Amikor begépelünk egy utasítást a kódszerkesztőbe, a Delphi a Turbo Pascalnál bevezetett és az ott jól bevált módszert használja annak elemzésére és feldolgozására. Ez a gépelés ideje alatt történik. Az Object Pascal kulcsszavak vastagon vannak szedve, a megjegyzések formátuma az Environment Options dialógusablak Colors oldalán található (és megváltoztatható) beállításokat követi.

A szóközöket és üres sorokat, tabulátorokat (közös nevükön: üres helyeket), kivéve ha azok idézőjelek között vannak, úgy kezeli a Delphi, mintha azok ott sem lennének. A kis- és nagybetűvel írott szavak között a Delphi nem tesz különbséget. A jobb áttekinthetőség kedvéért célszerű egységes formátumot használni programjainkban.

Az utasítások elé a Dephiben nem teszünk sorszámot, és minden utasítás általában egy külön sort foglal el. A Delphi 16-ban egy sor maximumum 127 karakterből állhat, míg a 32-bites verzióban 1024-ből. Egy utasítást – feltéve, hogy az szintaktikailag helyes – több sorra is bonthatunk az ENTER segítségével. Két utasítást egymástól minden esetben pontosvesszővel (;) választunk el. Mivel a pontosvessző és nem a kocsi vissza vagy a szóköz karakter az elválasztó, ezért egy sorba több utasítást is sűríthetünk, ha közéjük pontosvesszőt helyezünk. Az utasítássorozatot ízlésünknek megfelelően széthúzhatjuk (szóköz és tabulátor használatával).

Megjegyzés: *Néha előfordulnak majd a könyvben olyan sorok, amelyek a Delphi kódszerkesztő egy sorába is elférnének, de a könyvben több nyomtatott sort is kitesznek. Mivel a Delphi utasításokat pontosvessző választja el egymástól, így a pontosvesszők megkeresésével ez az (ál)probléma áthidalható.*

Programunk utolsó „end" utasítását egy pont követi (olyan amilyet a mondat végén is használunk). Ez jelzi a programunk végét, azaz, hogy nincs több végrehajtandó utasítás.

Megjegyzés: A Delphi – követve a Pascal hagyományait – megkülönbözteti az egyszerű és az összetett utasításokat. Az egyszerű utasítás egy végrehajtható programsort jelent. Az összetett utasítás nem más, mint egy egymástól pontosvesszővel elválasztott utasításblokk, amelyet a ***begin*** *utasítás vezet be és az* ***end*** *utasítás zár le. (Az összetett utasítás utolsó end előtti utasítása után nem szükséges pontosvesszőt tennünk.)*

Az összetett utasítások legtöbbször a ciklusokban és az if-then konstrukciókban fordulnak elő. Ezekről a programozási elemekről a fejezet későbbi részében még részletesebben lesz szó.

Megjegyzések

A programkódba azért helyezünk el megjegyzéseket hogy elmagyarázzuk, mit csinál az adott sor vagy blokk. A megjegyzéseket a Delphi nem hajtja végre és nem dolgozza fel, így nem is foglalnak el helyet a lefordított kódban. Két módon is lehetséges a megjegyzések beillesztése. Az első, ha a megjegyzést kapcsos zárójel közé helyezzük, ezzel választva el a megjegyzést a kód többi részétől.

```
procedure TForm1.FormClick(Sender: TObject);
begin
  {Az eljárást leíró megjegyzések helye}
end;
```

Mindig törekedjünk a megfelelő sortörésre az átláthatóság biztosítása céljából.

A második módszernél a megjegyzés elejét egy jobb oldali kerek zárójel és egy csillag [(*] a végét pedig egy csillag és egy bal oldali kerek zárójel [*)] jelzi. A megjegyzés több sort is elfoglalhat.

```
procedure TForm1.FormClick(Sender: TObject);
begin
  (*Az eljárást leíró
   megjegyzések helye*)
end;
```

A sor végére is kerülhetnek megjegyzések. Például:

```
begin {Az eljárást leíró megjegyzések helye}
```

A fenti megjegyzést határoló jelpárok bármelyike között szereplő szöveget a fordító megjegyzésként kezeli. Így ha elfeledkezünk a megjegyzés bezárásáról, akkor a fordító a nyitó jel utáni kód egészét megjegyzésnek tekinti. A megjegyzések szisztematikus használatának a hibajavításban is vitathatatlan szerepe van.

Programok befejezése

Ha a fejlesztői környezetben futtatott programot kívánunk megállítani, a legkönnyebb (és a legjobb) módszer az ALT-F4 billentyűkombináció használata, vagy a rendszermenü Close pontjának választása, miután biztosítottuk, hogy a fókusz a fő formon legyen. A futás megszakítása után visszakerülünk az integrált fejlesztői környezetbe (az IDE-be). Ha az ALT-F4 nem vezet célra, akkor általában a Run | Program Reset (CTRL-F2) ponttal léphetünk ki a futó alkalmazásból.

Megjegyzés: *A Run | Program Reset választás nem biztos, hogy felszabadítja a projektünkhöz lefoglalt Windows erőforrásokat.*

Másrészt a Delphi egy Application.Terminate utasítás (vagy a régebbi, de még támogatott Halt eljárás) végrehajtása után a program leáll, és az összes Delphi által a projekt számára lefoglalt erőforrás felszabadul. Egy önálló program esetén az Application.Terminate utasítás (vagy a Halt eljárás) végrehajtása után a Delphi a projekt által kinyitott összes ablakot becsukja, és a programot eltávolítja a memóriából. Ha az IDE-ben indí-

tottuk a programot akkor oda is kerülünk vissza az alkalmazás bezárásakor.

Tipp: *A Halt eljárás egy opcionális kilépési kódot használ, amely hasznos lehet, ha azt szeretnénk, hogy programunk képes legyen adatokat szolgáltatni annak az alkalmazásnak, amelyből az el lett indítva. További információkért érdemes elolvasni a beépített help megfelelő részét.*

A Delphi programon belül tetszőleges számú Halt eljárást használhatunk, de nyilvánvalóan jobb programozói megoldást jelent, ha korlátozzuk azoknak az eljárásoknak a számát, amelyek a programot terminálják. Sok Delphi programozó a programok befejezését kizárólag a főform OnCloseQuery vagy az OnClose eseményéhez köti.

Megjegyzés: *A Delphi akkor is megszakítja a program futását, amikor az egy RunError eljárást hajt végre (amely a futási időben bekövetkezett hiba jelzésére szolgál – l. még 8. fejezet) vagy egy Exit eljárásba ütközik a fő programblokkban.*

■ *Tulajdonságok beállítása*

A tulajdonságok programon keresztüli beállítása az egyik leggyakoribb feladat a Delphiben. A Delphi egy definiáló egyenlőségjelet (:=) használ erre a célra. Ha egy Delphibeli objektum tulajdonságának értékét meg szeretnénk változtatni, akkor először az objektum nevét, majd attól egy ponttal elválasztva a tulajdonság nevét kell feltüntetni, amit a definiáló egyenlőségjel és az új érték kell hogy kövessen:

objektum.tulajdonság := érték

Tegyük fel, hogy van egy Edit1 nevű TEditBox komponensünk, amelynek tartalmát törölni inkább a programban szeretnénk, mint az objektumfelügyelőn keresztül. Akkor a következő sort kell végrehajttatni a Delphivel:

```
Edit1.Text :='';
```

Az idézőjelek között nincs semmi, ami azt jelenti, hogy a tulajdonságnak üres szöveget adunk értékül. Hasonlóképpen az alábbi programsor:

```
Edit1.Text := 'Ez az új szöveg';
```

A Text tulajdonság az idézőjelek közt található szöveget kapja értékül. Programkód segítségével annyiszor változtathatjuk meg egy tulajdonság értékét, ahányszor arra csak szükségünk van.

Tervezési idő alatt megadott tulajdonságértékeket a programban megfelelő értékadásokkal írhatjuk felül. Például, ha azt akarjuk, hogy a Button5 nevű TButton komponens a fókuszátadási sorrendben (tab order) az első legyen, akkor illesszük a következő sort az egyik eseménykezelőbe:

```
Button5.TabOrder := 0;
```

Nézzük most meg, hogyan érhetjük el, hogy a formunk egy nyomógomb lenyomására elmozduljon. A form neve legyen Form1, a gombé MoveToLeft. Az alábbi eseménykezelő valósítja meg ennek a nyomógombnak az aktivizálását:

```
procedure TForm1. MoveToLeftClick (Sender: TObject);
begin
  Form1.Left := Form1.Left -75;
end;
```

Vizsgáljuk meg a program kulcssorát.

Form1.Left := Form1.Left – 75

Az értékadó utasítás bal oldalán található a tulajdonság, amely az értéket kapja, de furcsamód ez előfordul a jobb oldalon is. A sor a következőképpen kerül feldolgozásra: a Delphi – mielőtt az értékadást elvégezné – a definiáló egyenlőségjel jobb oldalát értékeli ki, meghatározva annak az értékét. Ebben az esetben a form (Form1) bal oldalának pozí-

ciójáért felelős Left tulajdonság aktuális értékét veszi elő, majd a kapott számból kivon 75-öt, és csak ezután nézi meg, hogy mi áll a sor bal oldalán. A bal oldali tulajdonság – jelen esetben a Left tulajdonság – megkapja a jobb oldal előbbiek alapján kiszámolt értékét, azaz a tulajdonság korábbi értékének függvényében változik meg.

A Delphi tulajdonságok közül sok maga is objektum. Erre egy jó példa a Font tulajdonság, amellyel több Delphi komponens is rendelkezik. A Font tulajdonság egyben objektum is (TFont típusú), ezért manipulálásakor ennek megfelelő szintaxist kell hogy használjunk. Az alábbi példa bemutatja, hogyan változtathatjuk meg egy szerkesztési ablakban használt fontméretet 24 pontosra:

```
Edit1.Font.Size := 24
```

Általában pontokat (.) használunk, ha lefelé kívánunk menni az objektumhierarchiában. (A 7. fejezetben lesz részletesen szó az objektumokról és azok hierarchiájáról.)

Logikai tulajdonságok

Azokat a tulajdonságokat, amelyek csak a True vagy a False (igaz/hamis) értéket vehetik fel, logikai vagy Boolean-típusú (George Boole angol logikus és matematikus után) tulajdonságoknak nevezik. Már sok logikai tulajdonságról ejtettünk szót. A Boolean-típusú tulajdonságok olyan jellemzőkért felelősek például, hogy egy nyomógomb látható-e, elérhető-e, az ENTER vagy az ESC billentyű lenyomásával aktivizálható-e a hozzá tartozó OnClick eseménykezelő (Default és Cancel tulajdonságok). Amikor a Delphi az alábbi utasítást végrehajtja, a nyomógomb láthatatlanná válik.

```
Button1.Visible  := False;
```

A komponens addig rejtve marad, amíg a Delphi végre nem hajtja a

```
Button1.Visible := True;
```

utasítást. Ha azt akarjuk, hogy egy adott komponens ne kaphassa meg az inputfókuszt a program futása közben, akkor állítsuk False-ra a komponens TabStop logikai tulajdonságát.

```
Edit1.TabStop := False;
```

Logikai tulajdonságok értékének átbillentését (True-ról False-ra ill. False-ról True-ra állítását) a Not művelet segítségével tehetjük meg. Az alábbi sor a Button1 nyomógomb láthatóságát változtatja meg:

```
Button1.Visible := Not (Button1.Visible);
```

Ez az utasítás a következőképpen kerül feldolgozásra: a Delphi megkeresi a Button1.Visible aktuális értékét, amelyet a Not művelet hatásaként negál, azaz ha értéke True volt, akkor False-ra változtatja és viszont.

A with kulcsszó

Ha egy objektum több tulajdonságát kell egyszerre megváltoztatni, valószínűleg mindenki szívesen elkerülné, hogy minden egyes tulajdonság elé oda kelljen biggyeszteni ugyanazt az objektumnevet. A **with** kulcsszó segítségével ez a nyűg kiküszöbölhető, azaz megkíméljük magunkat egy kevés gépeléstől. Ráadásul a **with** használata felgyorsítja a **with** blokkban található utasítások végrehajtását. Példaként tekintsük a DataInput nevű TEdit komponens magasságának, szélességének és színének megváltoztatását végrehajtó kódrészletet:

```
DataInput.Color := clWhite;
DataInput.Enabled := True;
DataInput.Height := 36;
DataInput.Font.Size := 24;
```

De gyorsabb és tisztább (és még kevesebb gépeléssel is jár) az alábbi kód használata:

```
with DataInput do
begin {Figyeljük meg a begin/end utasítászárójel
használatát!}
  Color := clWhite;
  Enabled := True;
  Height := 36;
  Font.Size := 24;
end;
```

■ *Változótípusok*

A Delphi mindkét verziója körülbelül 15 standard változó típus használatát támogatja. A Delphi-32 az előbbieket még (a számolás módjától függően) hét új típussal bővíti. A Delphi adattípusait különböző kategóriákra oszthatjuk fel: Boolean, Character, Integer, Real stb. (Lehetőség van saját típusok definiálására is, miként azt majd a következő fejezetben láthatjuk.) Az elkövetkezendő néhány oldalon keresztül a mindkét Delphi változatban megtalálható típusok rövid leírását ismertetjük. Ezt követi azon típusok bemutatása, amelyeket kizárólag a Delphi-32 támogat.

Megjegyzés: *Néhány nyelvben, mint pl. a Visual Basic, minden típus rendelkezik alapértékkel. Ez a Delphi típusok esetében nem igaz. Mindenképpen szükség van a kezdeti értékek megadására, különben kiszámíthatatlan következményekkel számolhatunk. (Az egyedüli kivételt ez alól az objektumok jelentik (l. 7. fejezet). Itt az inicializálás automatikusan megtörténik, erre viszont a jó programozó nem hagyatkozhat!)*

Boolean

Boolean (logikai) változók a True (igaz) és False (hamis) értékeket vehetik fel. Ennek a típusnak a memóriaigénye 1 byte.

Megjegyzés: A Delphi lehetővé teszi a ByteBool, WordBool és a LongBool logikai típusok használatát is. Ezeknek azonban nagyobb a memóriaigényük is. Az ilyen típusú logikai változók akkor tekinthetők igaznak, ha értékük 0-tól különböző. A Delphi programozók zöme nem használja ezeket a típusokat. Létezésüket a hamarosan bemutatásra kerülő Byte, Longint és Word típusokkal való kompatibilitás magyarázza.

Egész típusok

Az egész típusokat akkor használjuk, amikor egész számokkal dolgozunk (vagy érdektelenek a tizedespontot követő számjegyek). Az egész típusú adatokkal végzett műveletek rendkívül gyorsak. Hét típus tartozik az egészek közé, amelyek közül az öt legfontosabbat mutatjuk most be. (A Delphi 32-ben található még a SmallInt és a Cardinal típus, de ezeket viszonylag ritkán használjuk.)

Integer. Az Integer típusú változoknak nincs törtrészük. Ide viszonylag kis abszolútértékű egész számok tartoznak (–32 768 és 32 767 a Delphi 16-ban). Egy Integer típusú változó két byte-on tárolódik a Delphi 16-ban. A Delphi 32-ben ez egy 32 bites (4 byte-os), azaz. –2 147 483 648 és 2 147 483 647 közötti érték.

ShortInt. A ShortInt típusú változóknak nincs törtrészük, és viszonylag kis abszolútértékű egész számokat tárolnak (–128 és 127 között). A ShortInt típusú változók1 byte-ot foglalnak el a memóriából.

LongInt. A LongInt típusú változóknak nincs törtrészük, és viszonylag nagy abszolútértékű egész számokat tárolnak (–2 147 483 648 és 2 147 483 647 között) A LongInt típusú változók 4 byte-ot foglalnak el a memóriából. (A Delphi 32 Integer típusa a Delphi 16 LongInt típusával egyezik meg.)

Byte. A Byte típusú változók sem rendelkeznek törtrésszel, és nem negatív egész számokat tárolnak (0 és 255 között). Az ilyen típusú változók 1 byte-ot foglalnak el.

Word. A Word típusú változóknak nincs törtrészük, és 0-tól 65 535-ig terjedő egész számokat tárolnak. A Word típusú változók 2 byte-nyi memóriát foglalnak el.

Valós (lebegőpontos) számtípusok

Ezen típusok valamelyikét akkor használjuk, amikor nagyon kicsi ill. nagyon nagy (általában tizedes vesszőt is tartalmazó) számokra van szükségünk, de nem törekszünk a teljes pontosságra. A valós típusok mindegyikére igaz, hogy a velük végzett műveletek csak közelítések. A valós típusok műveletei jóval lassabbak, mint az egészeken végzett hasonló műveletek.

Megjegyzés: *Ezek az adattípusok csak akkor elérhetőek, ha a lebegőpontos (floating-point) fordítási direktíva (l. a 2. és a 3. fejezetet) be van kapcsolva. Mivel a Delphi azokat a programokat, amelyek a Visual Component Library-t (Vizuális Alkotó Elemek Könyvtárát) használják, nem tudja lefordítani, ha ez a direktíva ki van kapcsolva, a legjobb, ha nem változtatunk az alapbeállításon!*

A valósak közé az alábbi öt típus tartozik (ezen típusok egymásba konvertálhatóságáról l. később a Típus- és értékadás-kompatibilitás című alfejezetet):

Real. A Real típusú változók olyan törtrészt is tartalmazó számokat tárolnak, amelyek max. 11-12 értékes jeggyel rendelkeznek. A Real típusú változók 6 byte-nyi memóriát foglalnak el.

Megjegyzés: *Ez a típus kompatibilitási célokat szolgál. Használatát kerüljük! A 6 byte-on tárolt típusok nem szabványosak, ezért nem használhatjuk azokat, amikor más nyelvek által generált kóddal is dolgozunk.*

Single. A Single típusú változók olyan törtrészt is tartalmazó számokat tárolnak, amelyek max. 7-8 értékes jeggyel rendelkeznek. A Single típusú változók 4 byte-ot foglalnak el. Ez a típus csak akkor elérhető, ha a

lebegőpontos direktíva N+ állapotban van. Ez az alapbeállítás is, és a legtöbb esetben ebben az állapotban kell hogy legyen a helyes fordítás érdekében.

Double. A Double típusú változók olyan törtrészt is tartalmazó számokat tárolnak, amelyek max. 15-16 értékes jeggyel rendelkeznek. A Double típusú változók 8 byte-on tárolódnak. Ez a típus is csak akkor érhető el, ha a lebegőpontos direktíva N+ állapotban van.

Extended. Az Extended típusú változók olyan törtrésszel is rendelkező számokat tárolnak, amelyek max. 19-20 értékes jeggyel rendelkeznek. Az Extended típusú változók 10 byte-on helyezkednek el. Ez a típus is csak akkor elérhető, ha a lebegőpontos direktíva N+ állapotban van.

Comp. A Comp típusú változók nem a megszokott módon működnek, bennük rendkívül nagy méretű egészek tárolhatók (a törtrész levágása után). A Comp változók 8 byte-ot foglalnak el. Ez a típus is csak akkor elérhető, ha a lebegőpontos direktíva N+ állapotban van.

A Char és a String típus

Ezek a szövegkezeléssel kapcsolatos, előre definiált adattípusok a Delphi mindkét változatában szerepelnek. Segítségükkel lehetőség van különálló karakterek vagy max. 255 elemből álló karakterlánc kezelésére.

Char. A 16 bites Delphi verzióban és jelenleg még a Delphi 32-ben is a Char típusú változók egy byte-ot használnak egy darab ANSII/ASCII karakter tárolására. A Borland cég azonban kijelentette, hogy a Delphi 32 további változataiban a Char típus lehet, hogy Unicode karaktereket kezel majd. (Az Unicode egy olyan rendszer, amely 2 byte-ot használ a karakterek tárolására. Így több, mint 65 000 különböző karakter kezelésére nyílik lehetőség. Ha a Unikód karakter felső byte-ja 0, akkor az az alsónak megfelelő ANSI karaktert jelent.)

String. A String típusú változók a Delphi 16-ban egy 255 elemű karakterláncot tárolnak. A Delphi 32-ben azonban, ha másként nem állítjuk be, a karakterláncok méreteinek gyakorlatilag nincs megszabott határa.

(Valójában a határ 1 gigabyte, azaz 2^{30} byte.) A Delphi mindkét változatában lehetőség van a sztringek max. hosszának – feltéve, hogy ez 255-nél kisebb – korlátozására. A felső korlátot szögletes zárójelek közé téve szabhatjuk meg. Például:

```
var
  Name: String [45];
```

Mutatótípusok

A 6. fejezetben fogjuk mélyrehatóbban tanulmányozni az egyes mutatótípusokat. Azok, akik még nem programoztak mutatókkal, elégedjenek meg a következő definícióval: a mutatók adott objektumok memóriacímeit tárolják. A Delphi mindkét változatában két mutatótípus szerepel: a Pointer és a PChar.

Pointer. A Pointer típusú változók egy megadott típusú változóra mutatnak (az ún. alaptípusúra). Ezen kívül mutathatnak eljárások helyére is. Általánosságban azt mondhatjuk, hogy a mutató típusú változók változók memóriacímét tartalmazzák. L. még a 6. fejezetet.

PChar. Ez a mutató egy nulla-végű sztringre mutat. A nulla-végű sztringeket Windows rendszerrutinok meghívásakor és néhány szövegkezelő függvény megírásának egyszerüsítéséhez használjuk. A PChar nagyméretű sztringek létrehozásához és kezeléséhez is nélkülözhetetlen a Delphi 16-ban. L. még a 6. fejezetet.

A Delphi 32 új típusai

A Delphi 32-t néhány új típussal bővítették ki, amely nagyban megnöveli ütőképességét és igényeinkhez való alkalmazkodóképességét, ennek viszont az az ára, hogy az új típusokat használó projektek nem maradnak kompatíbilisek a Delphi 16-tal. Most ezeknek a típusoknak az ismertetése következik.

A Delphi 32 karaktertípusai

A Windows 95 és a Windows NT – az Unicode rendszer által – a nagyobb betűkészlettel rendelkező nyelvek használatát is támogatja (beleértve a kínaihoz hasonló ideografikus nyelveket). A csak 256 karakter megkülönböztetésére alkalmas 1 byte-os tárolás helyett, a Unicode 2 byte-on tárolja a karaktereket, ami több, mint 65 000 különböző karaktert jelent. Az ANSI karakterkészletnek azok a Unicode karakterek felelnek meg, amelyeknek első (a magasabb helyiértékű) byte-ja 0.

Az alábbi táblázat összefoglalja az új karaktertípusokat:

AnsiChar	Ez a Delphi 16 Char típusának felel meg. Ez a standard 1 byte-os (8 bites) ANSI karakter.
WideChar	Ez a 16 bites Unicode karakter típus.

Megjegyzés: A Delphi 32 jelenlegi változatában a Char típus a Delphi 16 Char típusának felel meg, mégsem ajánlanám a Char használatát a Delphi 32-ben írt alkalmazásokban, mert a Borland cég szerint elképzelhető, hogy a Delphi 32 elkövetkezendő változataiban a Char meg fog egyezni a WideChar típussal. Ha valamilyen okból mégis a Char mellett döntenénk, használjuk a beépített SizeOf függvényt a Char méretének meghatározására.

Új karaktermutató típusok

Az új karaktertípusok bevezetése kihatott a PChar típus sorsára is, amely szintén több részre hasadt. Ezeket a következő táblázat tartalmazza:

Mutató típusa	*Melyik karaktertípusra mutat*
PAnsiChar	AnsiChar
PWideChar	WideChar
PChar	Char

Új karakterlánc típusok

A Delphi 16 gyengéi közé tartozik, hogy a 255 karakternél hosszabb sztringek használata nem kevés plusszmunkát és odafigyelést igényel (a memóriafoglalást és -felszabadítást nekünk kell végeznünk). Méretük így sem haladhatja meg a 65 536 karaktert. A Delphi 32 teljesen kiküszöbölte ezt a problémát a hosszú karakterlánctípus bevezetésével.

Megjegyzés: A Delphi 32-ben a String-típus egy nagy ANSII karakterlánctípusnak felel meg. Ezt a fordítási direktívákon keresztül változtathatjuk meg (l. 6. fejezet).

Az alábbi táblázat a Delphi 32 új sztringtípusait foglalja össze:

Sztringtípus	*Jellemzői*
ShortString	Megegyezik a Delphi 16 String típusával. Max. 255 ANSII karakterből áll.
AnsiString	Lényegében korlátlan hosszúságú (1 gigabyte) ANSII karakterekből álló sztring. A Delphi 32 String és típusa alapértelmezés szerint ugyanaz.

Megjegyzés: Minden Delphi 32 komponens értelemszerűen támogatja az AnsiString típust.

Ahol lehet – azt ajánlom – használjuk a hosszú sztringtípust. A memóriafoglalás ekkor automatikusan megtörténik (referenciaszámlálással). Mivel a Delphi 32 referenciaszámlálást használ, a hosszú sztringek másolása tulajdonképpen gyorsabb, mint rövid sztringek esetében! Ráadásul a hosszú sztringek teljes mértékben kompatíbilisek a nulla-végű C sztring típusokkal, amelyek szükségesek a Windows belső függvényeinek a használatánál. (Meg kell tennünk viszont a megfelelő típuskonverziót a PChar és a PAnsiChar-hoz L. még a 6. fejezet típuskonverziókról szóló részét.)

A Currency típus

A Currency típust kimondottan pénzösszegekkel való számolásra alkották meg, segítségével a pénzösszegek „fillér" részét kerekítési hibák nélkül kezelhetjük (még milliárdos nagyságrendeknél is). Ez az új lebegőpontos típus egy skálázott 64 bites érték négy tizedesjeggyel, ami biztosítja a műveletek pontosságát.

A Variant típus

A Pascalhoz szokott olvasók számára első látásra idegennek tűnhet ez a típus. A Variant változókban az összes Delphi 32-beli adattípus tárolható. Nem számít, hogy a Variant típusban tárolt adat numerikus, dátum/idő vagy sztringtípusú. A Delphi automatikusan végrehajtja a szükséges konverziókat, így nem nagyon kell aggódnunk amiatt, hogy milyen adatfajtát adunk értékül egy Variant változónak. Többek között lehetőség van egy Variant változóban tárolt valós típusú és egy ugyancsak Variant változóban tárolt egész gond nélküli összeadására. (Sajnálatos módon viszont megnehezíti a hibakeresést, hogy numerikus értékeket hozzáadhatunk pl. sztringekhez anélkül, hogy hibaüzenetet kapnánk.)

A Variant változók használata lassabb mint más típusoké, az értékadásoknál és a különböző műveleteknél szükséges konverziók elvégzése miatt. A nagyobb memóriaigény (változónként 16 byte) sem a pozitívumai közé tartozik. Sokak szerint az automatikus konverziókra hagyatkozás hanyag programozói stílushoz vezet, amelyből nem sok előny származik. Ennek az egyik oka az, hogy gyakran a gép által végzett konverzióknak a programjainkban furcsa következményei lehetnek (kiderülhet, hogy az a konverzió, amelyet a Delphi végez nem egyezik meg a programozó által feltételezettel).

Megjegyzés: A könyvben csak akkor használjuk a Variant adattípust, amikor annak speciális tulajdonságaira van szükségünk. Ez megfelel annak a programozási elvnek, hogy a programozónak meg kell őríznie a programja feletti kontrolt, és csak akkor hagyatkozzon az automatikus konverziókra, ha teljes mértékben tudatában van annak következményeivel. Ha a Variant típust szeretnénk használni, célszerű megismerkednünk előbb a VarType függvénnyel a beépített helpben.

Ennek a függvénynek a segítségével meghatározhatjuk a Variant típusú változókban tárolt érték jellegét.

Amikor mégis szükség van a Variant használatára (pl. OLE Automation használatakor l. 13. fejezet), gyakran meg kell változtatnunk a benne tárolt adatok típusát. Ezt a 6. fejezetben ismertetendő konverziós technika (*typecasting*) segítségével tehetünk meg.

■ *Változók*

A változók a rugalmas információ(érték)-hordozást valósítják meg a Delphiben és más programnyelvekben is. Ellentétben a Visual Basic-ben megszokottakkal, a Delphiben minden változót deklarálnunk kell, mielőtt azokat használnánk. A deklarációkat a **var** kulcsszó után tehetjük meg. Itt kell megadni a változó nevét és típusát, egymástól kettősponttal elválasztva. Például:

```
var
  Kamat: Double;
  Méret: Integer;
  NagyEgész: LongInt;
  HanyagVáltozó: Variant;     {Csak Delphi 32-ben}
```

Ugyanolyan típusú változókat vesszővel elválasztva egyszerre is deklarálhatunk (nem úgy, mint Visual Basic-ben). Például:

```
var
  Kamat, ForgalmiAdó: Double;
```

A Delphiben a változónevek akármilyen hosszúak lehetnek, de csak az első 63 karakter szolgál azonosításukra. Az ezek után álló karaktereket a fordító figyelmen kívül hagyja. Az első karakternek egy betűnek vagy aláhúzásjelnek (_) kell lennie. A változónév számok, betűk, aláhúzások bármilyen kombinációja lehet, de nem tartalmazhatnak olyan jeleket, mint a $, % stb. Nem használhatjuk változóként a fenntartott szavakat. A változókban használt betűk kicsi vagy nagy volta nem jelent különbséget.

Megjegyzés: Ezek valójában az azonosítókra vonatkozó szabályok, amelyeket már láttuk korábban is, hiszen a Delphi komponensek Name tulajdonságában is azonosítókat használunk.

Az alábbi táblázatban felsorolunk néhány változónevet. Külön jelezzük azokat, amelyeket nem használhatunk a Delphiben.

Base1_Ball	használható
$Base_Ball	nem használható, mert az első karakter nem betű, szám vagy aláhúzás
Base.1	nem használható, mert pontot tartalmaz
EzHosszúDeJó	használható és rövidebb, mint 63 karakter, tehát minden betű számít

Ugyan az első 63 karakter különbözteti meg a változókat, de az, hogy nagy- vagy kis- betüket használunk-e, nem számít. A *BASE* nevű változó ugyanaz, mint a *base.* Ugyanakkor a *Base* és a *Base1* különböző változókat jelöl, és a *Base_1* név pedig egy harmadiknak felel meg.

Úgy mint az objektumnevek körül, itt is parázs vita robbant ki arról, hogy milyen legyen a szabványos változónév. Például sokan a Microsoft „magyar" konvencióját használják (néhányan még kedvelik is), aholis a névben a változó típusát is megjelölik egy kisbetűs előtag formájában. Például az iMérték névben az i az egész (Integer) típust jelzi. A vitától távol tartva magam, csak azt a minden programozó számára nyilvánvaló tényt szeretném hangsúlyozni, hogy a jelentéssel bíró elnevezések nagy mértékben hozzájárulnak a programok áttekinthetővé tételéhez és segítenek a hibakeresésben is.

Megjegyzés: A könyvben nem a magyar konvenciót követjük, egyszerűen csak jelentéssel bíró részekből álló változóneveket használunk, amelyek értelmes részeit nagy kezdőbetűvel írjuk (például: KamatosKamat és nem Kamatoskamat). (Ezt a konvenciót vegyes változónév-használatnak nevezik, ami nem túl elterjedt a régebbi stílusú Pascal programozásban.)

A Delphi fenntartott szavait nem használhatjuk változónevekként. Például a Do nevet nem adhatjuk egyik változónknak sem. Nem szabálytalan azonban, ha a változó neve tartalmaz fenntartott szót. Például a DoIt-ot nyugodtan használhatjuk. Ha a programban mégis előfordulna olyan vál-

tozónév, amely egyben fenntartott szó is, akkor annak jelenlétét a Delphi – általában fordítás közben – hibaüzenettel jelzi. (A beépített súgó felsorolja a fenntartott szavakat.)

Értékadás változóknak

Épp úgy, mint a tulajdonságok esetében a Delphiben egy kettőspont és egy egyenlőség-jel (definiáló egyenlőségjel :=) használatával adhatunk értéket a változóknak. Például:

```
procedure TForm1.FormClick(Sender: TObject);
var
  InterestRate: Double;
  TextEntry: String;
begin
  InterestRate := 0.05;
  TextEntry := Edit1.Text;
end;
```

A változónév mindig a definiáló egyenlőség bal oldalán jelenik meg, az érték pedig a jobb oldalon. A Delphinek minden esetben először a jobb oldali értékkel kell foglalkoznia, és mindent meg kell hogy tegyen ennek az értéknek a kiszámolásához. A példában szereplő második értékadás megkeresi a szerkesztési ablak Text tulajdonságának értékét, majd bemásolja a TextEntry változóba. Az értékadó utasítás nem más, mint egy adatmásolási eszköz, tudniillik érték adódik át a jobb oldali kifejezésből a bal oldali változónak.

Típus- és értékadás-kompatíbilitás

A Delphi a Pascalból nőtte ki magát, ennek következtében itt is szigorú szabályok vonatkoznak arra (kivéve ha a Delphi 32 Variant típusát használjuk), hogy a változótípusok között milyen értékadás végezhető el. Például a program, amelyben az alábbi rész szerepel, fordításkor egy „Type Mismatch" („A típusok ütköznek") hibaüzenettel áll le.

```
var
  EgészVáltozó: Integer;
begin
  Egészváltozó := 3.5
```

Megjegyzés: A programozók egy része nem lelkesedik az ilyen merev típushasználati lehetőségért. A Delphi 16-ban eleve nem sokat tehetünk ez ellen, és a Delphi 32-ben sem javaslom a Variant típus használatát a probléma áthidalására. A Variant változók nemcsak termőtalajul szolgálnak a hibáknak, hanem jóval lassabbak is, és több memóriát használnak, mint a szigorúan ellenőrzött típusok.

Két Delphi alaptípus kompatíbilis, ha:
- mindkettő valós,
- mindkettő egész,
- az egyik karakter a másik sztring,
- mindkettő sztring típusú.

Ahhoz azonban, hogy egy érvényes értékadást adjunk meg – a típusok kompatibilitása mellett – szükséges, hogy az aktuális értékek is kompatíbilisek legyenek. Így egy Integer típusú változóhoz hozzáadhatjuk – kivételkezelés használata nélkül – azt a LongInt típusú változót, amelynek aktuális értéke pl. 123. (A 8. fejezetben lesz szó az itt említett kivételkezelésről.)

■ *Konstansok*

A hibajavítást akkor a legkönnyebb. elvégeznünk, ha a programunk olvasható. Próbáljuk meg elkerülni a SZEF („a szemem elsiklott fölötte") szindrómát, amely nagyon gyakori, ha a programunk sok elszórt rejtélyes számot tartalmaz. Sokkal könnyebb átlátni és karbantartani egy olyan programot, amelyben jelentéssel bíró nevek szerepelnek a semmitmondó számok helyett. Ezt konstansok alkalmazásával érhetjük el. Például a Tax Rate (adókulcs) nevű konstans bevezetésével egyszerűbben érthetjük meg az alábbi kódrészletet

```
TotalTax := Konst*TAX_RATE;
```

mint,

```
TotalTax = Konst*0.0825;
```

Konstansok bevezetésével könnyen memorizálható nevekkel illethetünk olyan értékeket, amelyek nem változnak a projekt futása alatt, és így a számok megjegyzése helyett elegendő az azokat jelölő név használata. (Lehetőség van azonban a konstansok értékének megváltoztatására két fordítás között anélkül, hogy az összes számot meg kellene változtatni. Ez a jellemző nagyban növeli a program karbantarthatóságát.)

Habár a Delphiben nem előírás (a szabványos Pascallal ellentétben), mégis a legtöbb esetben a konstansok deklarációja a változódeklarációk előtti szakaszban történik. A sornak, amelyikben a konstansnak értéket adunk, tartalmaznia kell a konstans nevét, egy egyenlőségjelet és a kívánt értéket.

```
const
  FORGALMI_ADÓ = 0.0825;
  NYELV_NÉV = 'Delphi';
var
{A változódeklarációk helye}
```

A konstansoknak – úgy mint a változóknak – van típusa, de a konstansok a deklarációban megadott értékből származtatják ezt a típust. Például a FORGALMI_ADÓ típusa Single vagy Double (a megadott számtól függően), a NYELV_NÉV pedig nyilván sztring.

A konstansnevek is Delphi azonosítók, ezért a rájuk vonatkozó szabályok megegyeznek a változóknál ismertetettekkel: csak az első 63 karakternek van valódi szerepe, az első karakternek betűnek vagy aláhúzásnak kell lennie, amelyet betűk, aláhúzások és számok bármilyen kombinációja követhet. Ahogy a példákból kitűnt, a könyvben a programozó által definiált konstansokat nagybetűvel írjuk, az értelmes (de nem egybe írandó) szavak közé pedig aláhúzást teszünk.

Bizonyos esetekben lehetőség van egy konstans másik konstanssal és/vagy Delphibeli függvénnyel történő definíciójára. Például:

```
const
  KAMAT = 0.18;
  NAPI_KAMAT = KAMAT/365;
```

(Azt, hogy mely függvények használhatók ilyen esetben, a beépített helpből tudhatjuk meg.)

Megjegyzés: A Delphi többszáz beépített konstanssal rendelkezik, amelyek bizonyos függvényekben és objektumokban jól használhatók.

Az előző fejezetben már használtunk az előre definiált konstansok közül néhányat, amikor tulajdonságokat állítottunk be fejlesztési időben. Ezek a konstansok azokban a unitokban találhatók, amelyek megvalósítják az adott objektumot vagy függvényt. A beépített konstansokat nem kell deklarálni. Ezekről bővebbet az adott objektumról vagy függvényről (amellyel éppen dolgozunk) szóló fejezetben találhatunk a beépített súgóban. Példaként tekintsük a következő utasítást:

```
Form1.Color = clBlack;
```

Ez a sor a form színét változtatja feketére a beépített clBlack konstans használatával. (A Delphi beépített konstansai általában – de nem mindig – két részből állnak: egy kisbetűs előtagból, amely a tulajdonságra utal, és egy nagy betűvel kezdődő szóból, amelyből általában világosan kitűnik a konstans szerepe.)

■ *A Delphi eseménykezelők anatómiája*

Az előző fejezetekben már láttuk az összes olyan elemet, amelyekhez eseménykezelő. eljárások (vagy bármilyen célnak megfelelő Delphi eljárások – l. 6. fejezet) kapcsolhatók. Az eseménykezelőket egy fejléc vezeti be, amelyben az eljárás paraméterei vannak felsorolva, ezeket az eljárás későbbi része használja. (L. még a 7. fejezetet.) Ezt követi (nem kötelezően) egy **const** szekció a konstansok deklarálására, majd a **var** blokk, ahol a változókat deklaráljuk. Ezután a **begin** fenntartott szó kell hogy

következzen, amely a kód törzsének kezdetét jelzi. A kód törzsét az **end** utasítás és az azt követő pontosvessző zárja le. (A program legvégét egy **end** és egy pont jelzi.)

Változók és konstansok hatásköre

Ebben a fejezetben azt vizsgáljuk meg, hogy a program egyik felében deklarált változók és konstansok elérhetők-e a program egy másik részében lévő függvények és műveletek számára. Régebbi programozási nyelvekben – ahol minden változó és konstans elérhető volt a program bármelyik részében – előfordult, hogy a nevek keveredtek. Például egy korai, strukturálatlan nyelvben két Összeg nevű változó használata (ami tulajdonképpen egy) a program különböző részeiben sok fejfájás okozója lehetett, hiszen azok egymás értékeit írták felül.

Modern progamozási nyelvekben a megoldást a változók és konstansok elszigetelése jelenti. Így az egyik eljárásban szereplő Összeg nevű változó értékének megváltoztatása nem hat ki a másik eljárásban lévő ugyanilyen nevű változó értékére, hacsak másként nem írjuk elő. A Delphi változók lokálisak, hatáskörük – ha nem jelöljük meg másként – csak az őket tartalmazó eljárásra terjed ki. Az egyik eseménykezelő általában nem férhet hozzá a másik eseménykezelő változóihoz.

Megjegyzés: *Mivel a Delphi megköveteli a változók deklarálását, ezért sokkal ritkábban jönnek elő különböző mellékhatások.*

Nem túl jó gyakorlat vakon megbízni az alapbeállításokban. Ha biztosak akarunk lenni abban, hogy egy változó lokális, akkor azt az eseménykezelő **var** blokkjában fel kell tüntetni. A változót használat előtt deklarálni kell, ezért ha azt tapasztaljuk, hogy a deklarálatlan változóra is hivatkozni lehet, biztosak lehetünk abban, hogy egy másik eseménykezelőből származó, nagyobb hatáskörű változóval van dolgunk.

Eljárások közös változói

Természetesen elképzelhető, hogy a változók értékeit más eseménykezelők vagy unitok számára is elérhetővé kívánjuk tenni. Például ha egy alkalmazásban egyszerre több eljárásban végzünk számításokat ugyanazzal a kamatlábbal, akkor az a unit minden eljárása számára elérhető kell hogy legyen. Azokat a változókat, amelyekkel ezt meg lehet valósítani, *globális* változóknak nevezzük. A deklarálás helyétől függően, egy változó lehet tisztán globális, unit szintű, objektum szintű vagy eljárás szintű. A globális változók az alkalmazás bármely részében, a unit-szintű változók az adott unit minden részében elérhetők. Az objektum szintű változók a formhoz tartozó összes eseménykezelő által elérhetőek, míg az eljárás szintű változók hatásköre csak az adott eljárásra vagy eseménykezelőre terjed ki.

A unit szintű változók deklarációja a unit **implementation** (implementációs) blokkjának egyik **var** szekciójában történik. Az unit-szintű konstansok a unit **implementation** blokkjának **const** részében deklarálandók. Az implementációs blokk megkeresésének a módja a következő: nyissuk meg a kódszerkesztő ablakot, álljunk a form unit fájljának a tetejére, és lefelé haladva keressük meg az **implementation** fejléccel rendelkező részt. (További részletekért l. a 6. fejezetet.)

Az alábbi példa azt mutatja be, hogyan lehet a Kamatláb nevű változót és a FORGALMI_ADÓ nevű konstanst unit-szintűnek deklarálni:

```
implementation
{$R *.DFM} {Nem baj, ha ezeket még nem értjük, de ne
változtassuk meg őket.}
const
  FORGALMI_ADÓ = 0.0825
var
  Kamatláb: Comp;
```

Ezzel elértük, hogy

- a FORGALMI_ADÓ nevű konstans és a Kamatláb nevű változó a unit minden eljárása számára látható.
- ha az egyik eseménykezelőben megváltoztatjuk a változó értékét, akkor az a többi eseménykezelőben is érvényben marad.

Nyilvánvaló, hogy a második pont alapján óvatosaknak kell lennünk, az unit-szintű változók értékadásánál. Az eseménykezelők közt átadott adatok gyakori hibaforrásul szolgálnak. Ráadásul ezeket a hibákat meglehetősen nehéz lokalizálni.

Lehetőség van arra – habár sok programozó kerüli –, hogy ugyanazt a változó- vagy konstansnevet egyidőben unit szintű és lokális változóként vagy konstansként is használjuk. Egy eljárás **const** és **var** blokkjában felsorolt konstansok és változók közül azok, amelyek globális konstansokként ill. változókként voltak deklarálva, felüldeklarálódnak. Ezek lokális változóként jelennek meg az adott eljárásban, ami azt jelenti, hogy az „előző életükben" hordozott információ ideiglenesen elvész – vagy programozói nyelven –, láthatatlanná válik az eljárás számára. A Delphi nem jelzi, ha egy globális változó nevét használva definiálunk egy lokális változót. Ez egy okkal több arra, hogy ne felejtsük el az eljárás belsejében deklarálni azokat a változókat, amelyeket lokálisként akarunk használni.

Tipp: *Vannak programozók, akik szeretik megjeleníteni a változók nevében azok hatáskörét is, például az esemény szintű változók nevét kis p-vel (az angol procedure-level szóból) az unit szintű változók nevét pedig kis u-val (az angol unitlevel szóból) kezdik. Ezzel első látásra meg tudjuk mondani a változók hatáskörét.*

Ahogy azt korábban láttuk, a konstansok is hasonlóan viselkednek: ha csak egy formunk van, vagy azt szeretnénk, hogy a konstansok csak egy form eseménykezelői számára legyenek láthatóak, akkor a unit implementációs blokkjában helyezzük el őket. Ha egy eljáráson belül definiálunk egy konstanst, akkor az csak az eljáráson belül látszik. (L. még később a több formot használó programokról szóló részben.)

Megjegyzés: *Ha globális konstansokat és változókat kívánunk létrehozni egy unitban (azaz olyanokat, amelyek ezt a modult használó minden unit számára elérhetőek), akkor azok deklarációját a unit* ***interface*** *blokkjában tegyük meg.*

Statikus változók használata

Amikor a Delphi meghív egy eseménykezelőt, a lokális változók régi értékei elvesznek, és visszakapják inicializált értékeiket. (Ahogy azt már korábban említettük, a változókat nem hagyhatjuk inicializálatlanul, mindegyiknek kezdőértéket kell adni.) Azokat a változókat, amelyek nem nyerik vissza értéküket *dinamikus* változóknak hívják. A dinamikus változók önmagukban nem elegendőek minden programozási szituáció kezeléséhez. Tegyük fel, hogy számon akarjuk tartani, hogy egy adott gomb hányszor volt lenyomva. Nem jutunk messzire, ha a számláló mindig lenullázódik. Ezt a problémát unit-szintű (form-szintű) változók használatával ugyan áthidalhatjuk, ezeket azonban csak akkor célszerű bevezetni, ha azt akarjuk, hogy más eseménykezelők is el tudják érni őket. Ennek a módszernek az alkalmazásával tehát a számlálás működne, de a számláló nem csak az őt használó programrészben lenne látható (ami fölöslegesen túlteljesítené kívánságunkat).

Amire itt ténylegesen szükségünk van, az az, amit a Visual Basic-ben és néhány más nyelvben *statikus* változónak hívnak. Ezt a változófajtát a Delphi a program futásának megkezdésekor inicializálja. Ezután értékét megtartja; ha az adott eljárás megváltoztatja, akkor a következő híváskor az eljárás ugyanazt az értéket látja. A Delphiben (amely a Pascaltól örökölte a terminológiát) ezeket a változókat *típusos konstansoknak* nevezik. Fontos tudni, hogy ezek nem konstansok – a megtévesztő nevet azért kapták, mert deklarálásuk a konstans blokkban történik, és ekkor kapják meg kezdeti értéküket is. Az alábbi szintaxist használjuk:

```
const
    számláló: Integer = 1;
```

Visual Basic Tipp: *Lehet, hogy az elnevezés nem igazán találó, de ezért kárpótol minket a Delphi típusos konstansainak egy nagy előnye a Visual Basic statikus változóival szemben: a típusos konstansok kezdeti értékének bármit megadhatunk.*

Most nézzük meg a típusos konstansokat (statikus változókat) működés közben. Először helyezzünk egy nyomógombot (TButton) egy üres formra, majd adjuk hozzá az alábbi OnClick eseménykezelő eljárásokat:

```
procedure TForm1.Button1Click(Sender: TObject);
const
  Számláló: Integer = 1;
  begin
    ShowMessage(IntToStr(Számláló));
    Számláló := Számláló + 1;     {Inc(Számláló) is
                                   jó ide}
  end;
```

Most próbáljuk ki a következő eljárást, amely egy inicializálatlan változót használ számlálóként:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Számláló: Integer;
  begin
    ShowMessage(IntToStr(Számláló));
    Count := Count+1
  end;
```

Megfigyelhetjük, hogy az utóbbi esetben meglehetősen furcsa számokat kapunk a Számláló értékére. Végül változtassuk meg ezt az eseménykezelőt úgy, hogy 1-et adunk kezdeti értékül a Számláló változónak. Azt tapasztaljuk, hogy akárhányszor futtatjuk le a programot, a gomb lenyomása után a Számláló értéke 1 lesz.

Tipp: *Számlálók készítése mellett a típusos konstansok (statikus változók) alkalmasak például komponensek láthatóságának ki- és bekapcsolására (vagy hasonlóképpen bármilyen más logikai tulajdonság értékének beállítására) is, továbbá hibakereső eszközként is bevethetők. A TTimer komponenst eseménykezelő eljárásában használva, nagyobb késleltetéssel működő időkapcsolót valósíthatunk meg. Az utóbbi esetben a típusos konstansban tartjuk számon, hogy hányszor lett az eljárás meghívva.*

■ *Karakterláncok (sztringek)*

A Delphi szerkesztési területeibe beírt információk tárolására karakterláncokat használunk, ez is jól mutatja nélkülözhetetlenségüket.

Megjegyzés: *A Delphi 16 sztring változóiban max. 255 karaktert tárolhatunk, hosszabb karakterláncok kezelését más technikákkal kell pótonunk. A 6. fejezetben megismerkedhetünk néhány ilyen módszerrel.*

Két sztringet a plusszjel (+) segítségével fűzhetünk össze (konkatenálhatunk). Például ha:

```
strCím := 'királynő';
strNév := 'Erzsébet ';
strSorszám := 'I';
strPont := '. ';
```

akkor

```
strSorszám + strPont + strNév + strCím =
'I. Erzsébet királynő'
strSorszám + strSorszám + strPont + strNév + strCím =
'II. Erzsébet királynő'
```

Tipp: *A Delphi 16 sorméret-korlátozásai miatt szükségünk lesz a + sztring műveletre, ha 127 eleműnél hosszabb karakterlánccal akarunk dolgozni. (És nyilvánvalóan elkerülhetetlen a + használata, ha ki szeretnénk használni a Delphi 32 1 gigabyte-os sztringméretét.)*

A + művelet a megadott sorrendben kapcsolja össze a karakterláncokat. Itt fontos a sorrend, nem úgy, mint számok összeadásánál. A + jel használatával új sztringet is definiálhatunk. Ekkor az összefűzött sztringet értékül adjuk az új sztringváltozónak. (L. alábbi példa, ahol a korábban definiált változókat használjuk.)

```
strTeljesNév := strSorszám + strSorszám + strPont +
strNév + strCím;
```

Tipp: *Ha egy sztring típusú tulajdonságot meg akarunk változtatni, és a változtatás több egymást követő műveletből áll, akkor sokkal jobban járunk (végrehajtási sebesség szempontjából), ha a műveleteket egy karakterlánc típusú változóval hajtjuk végre, és csak azután másoljuk be azt az adott tulajdonságba.*

Egy karaktert megadhatunk az ASCII kódjával is, ekkor a # operátort használjuk az ASCII kód előtt. Például

```
ABetű := #65;
```

megegyezik az

```
ABetű := 'A';
```

értékadással. Egy másik példa:

```
ÚjSor :=. #10 + #13 {Használhatjuk a #10#13-at is}
```

Az ASCII 10+ ASCII 13 az ÚjSor kódnak felel meg.

■ *Számok*

A nagy számok írásmódja eltér a magyar számírás szabályaitól: az ezres csoportok közé nem teszünk szóközt. A tizedesvessző helyett tizedespontot használunk. (Természetesen a tizedespont használata nem megengedett egész típusok esetén.) Ha egy változónak numerikus értéket kívánunk adni, akkor a szám a definiáló egyenlőség jobb oldalára kerül.

Megjegyzés: *A .5 formátum használata nem megengedett, mindenképpen tegyünk egy nullát a tizedespont elé (0.5).*

Ha egy tizedespontot tartalmazó számot próbálunk értékül adni egy egész típusú változónak, a fordítás alatt hibaüzenetet kapunk.

Megjegyzés: Az Integer, Shortint, Smallint, Byte, Word, Cardinal és a Longint típusú változók értékét hexadecimális alakban is megadhatjuk, ha a szám tizenhatos számrendszerbeli reprezentációja elé egy $ jelet teszünk.

Például a $A és a 10 (tizes számrendszerbeli) ugyanazt jelenti a fordító számára.

Megjegyzés: Programunk kiszámíthatatlanul viselkedik, ha egy változónak a típusára jellemző korlátokon kívül eső számot adunk értékül. Hibaüzenetet azonban csak akkor kapunk, ha a Range checking (értéktartomány-ellenőrzés) fordítási direktívát beállítjuk.

Ha azt akarjuk, hogy a Delphi értéktartomány-ellenőrzést végezzen, és így a típushatárokon kívüli értékekre a fordító hibaüzenetet adjon, akkor a Range checking fordítási direktívát be kell kapcsolnunk. A Delphi 32-ben ezt Project | Options pont választásával megjelenő dialógusablak Compiler oldalán találhatjuk. A Delphi 16-ban ugyanez a dialógusablak az Options | Project menüpont választásával nyitható meg. (L. még a 6. fejezet fordítási direktívákról szóló részét.) A direktíva bekapcsolásának megvan az ára: programunk futása lassabbá válik. Ezért érdemes csak tesztelésnél bekapcsolni.

Műveletek számokkal

Az alábbi táblázat bemutatja a. négy számtani alapművelethez tartozó műveletjeleket. Az osztás kivételével az alapműveletek mindegyikére igaz, hogy két azonos típusú számhoz ugyanolyan típusú számot rendel.

Delphibeli operátor	*Művelet*
+	Összeadás
-	Kivonás (és negatívszám-képzés)
/	Osztás (mindig valós típusú eredményhez vezet – még egész számok osztása esetén is)
*	Szorzás

Visual Basic tipp: *A beépített Exp függvény használatával kaphatjuk meg a Visual Basic hatványozást reprezentáló ^ művelet megfelelőjét.* ($a^b = exp(b*ln(a))$)

Az egész típusokhoz még két fontos számtani művelet tartozik (l. alábbi táblázat). Ezek csak egész típusú változókra alkalmazhatók.

Delphibeli operátor	*Művelet*
div	Egész osztás
mod	Maradékképzés

Ellentétben a / operátorral, amely valós típusú erdményhez vezet, a div operátor használatával egy adott egész típuson belül maradhatunk. A div a maradékot már nem osztja tovább, így adhat egész számot eredményül. Például: 7 div 2 = 3.

A mod operátor az egész osztás másik felét valósítja meg. Ezzel az operátorral az egész osztás utáni maradékot kaphatjuk meg. Például: 7 mod 3 = 1. Ha a jobb oldalon lévő számban pontosan megvan a bal oldalon található szám, akkor nincs maradék, azaz a mod művelet 0-t ad eredményül: 8 mod 4 = 0.

A számok, változók és műveletek olyan kombinációját, amelyet kiértékelve a Delphi valamilyen számértéket kap, *numerikus kifejezésnek* nevezzük.

Logikai és bináris operátorok

A Delphiben az 5-1. táblázatban bemutatott operátorok használatával kapcsolhatunk össze logikai változókat és kifejezéseket.

5-1. táblázat. Logikai műveletek

Szimbólum	*Függvény*
= (egyenlőség)	Teszteli, hogy mindkét logikai kifejezés igaz-e vagy hamis.
not	Logikai kifejezések értékét változtatja igazról hamisra és fordítva.
and	Logikai ÉS, akkor és csak akkor igaz (True), ha mindkét kifejezés értéke igaz.
or	Logikai VAGY, akkor és csak akkor igaz (True), ha legalább az egyik kifejezés értéke igaz.
xor	Logikai KIZÁRÓ VAGY, akkor és csak akkor igaz (True), ha a logikai kifejezések értéke különböző.

Ezek a műveletek egész számok esetén is használhatók. Ekkor azok az egész szám vagy számok egyes bitjeire hatnak. Például az **and** (logikai és) művelet pontosan akkor ad 1-et az eredmény aktuális bitjében, ha a műveletben szereplő mindkét szám bináris alakjában a megfelelő helyiértékű bit 1. Így:

X and 1	Megvizsgálja, hogy az X első bináris jegye 1-e.
X and 2	Megvizsgálja, hogy az X második bináris jegye 1-e.
X and 3	Megvizsgálja, hogy mind az első, mind a második bit 1-e.

(Ezt az eljárást gyakran *maszkolásnak* hívják, szükségünk lehet rá, amikor a Delphi vagy Windows API függvények visszatérési értékeit kell kiértékelnünk.)

Az 5-2. táblázat tartalmazza ezeket a bitenkénti műveleteket és azok rövid leírását.

A táblázatban felsoroltakon kívül létezik még két hasznos, egészekkel végezhető művelet. Az shl (eggyel balra eltol) és az shr (eggyel jobbra eltol) operátorok a szám bitjeit mozgatják eggyel balra illetve eggyel jobbra. Hatásuk tehát nem más, mint egy egész szám 2-vel való szorzása illetve kettővel való osztása, előnyük a gyorsaság. Több bitpozícióval is arrébb ugorhatunk, ha az shl vagy shr mellé még egy egész számot is írunk. Például az shl 3 művelet három bitpozícióval mozgatja balra a megadott számot, ami a 8-cal való szorzás egy gyors megvalósítása.

5-2. táblázat. Bitenkénti műveletek.

Bit-műveletet tartalmazó kifejezés	*Hatása*
X and Y	1-et ad abban a bitpozícióban, amelyikben X és Y bináris alakjában is egy áll, különben 0-át.
X or Y	1-et ad abban abban a bitpozícióban, amelyikben az X és az Y közül valamelyiknek 1 áll a bináris alakjában.
X xor Y	1-et ad abban a bitpozícióban, ahol az X és az Y közül pontossan az egyiknek áll 1 a bináris alakjában.
not (X)	X bitjeit negálja (1 helyett 0, 0 helyett 1).

Zárójelezés és műveleti sorrend

Két út áll előttünk, hogy a műveletek elvégzésekor követendő sorrendet megjelöljük. Az első a zárójelek használata, sok programozó ezt választja. A zárójeleket a matematikában szokásos módon használjuk, segítségükkel egyszerű a sorrend meghatározása. Például a 3 + (4 * 5) kifejezés 23-at ad eredményül, mert a Delphi először a zárójelen belüli kifejezést értékeli ki (4-szer 5) és csak azután adja hozzá a 3-at. A (3+4) * 5 kifejezés értéke 35, mert itt először a Delphi összeadja a 4-et a 3-mal, és csak azután szoroz 7-tel.

Tekintsünk most egy másik példát:

```
((6 * 5) + 4) * 3
```

A legbelső zárójelen belüli kifejezés 30-at ad, a második pár zárójelen belüli kifejezés a Delphit 4 hozzáadására utasítja, amelynek eredménye 34, és végül ez szorzódik meg 3-mal, ami 102-es végeredményhez vezet.

Lehetőség van egyes esetekben a zárójelezés elkerülésére is, feltéve hogy követjük a szabályt, amely meghatározza a matematikai műveletek sorrendjét. Például a szorzás megelőzi az összeadást. Ez azt jelenti, hogy a 3 + 4 * 5 kifejezés 23-mal egyenlő, és nem 35-tel, mivel a szorzást (4 * 5) a Delphi az összeadás előtt végzi el.

A következő lista a műveletek sorrendjét (hierarchiáját) mutatja:

```
@, not
*, /, div, mod, and, shl, shr, E
+, -, or, xor
=, <, >, <=, >=, <>, in
```

Az utolsó sor a relációkat tartalmazza. Ezeket a fejezet későbbi részében ismertetjük – kivéve az *in*-t, amelyet a halmazokkal használunk, erről a 6. fejezetben lesz szó.)

Amikor két egy szinten lévő operátort használunk, a Delphi balról jobbra haladva hajtja őket végre.

Néhány szó még a Delphi számhasználatáról

Ha az olvasó már próbált műveleteket végezni nagy számokkal a Delphiben, valószínűleg láthatta hogy a Delphi gyakran nem ír ki hosszú számokat, helyettük a számok normál alakját jeleníti meg. Azoknak, akik nem jártasak ebben a jelölésmódban, a következőket kell észben tartaniuk: az E+ azt jelenti, hogy a tizedespontot – az esetleges nullák hozzáírásával – mozgassuk jobbra. Azt, hogy hány hellyel lépjünk odébb, az E utáni szám jelöli. Ha az E után negatív szám áll, akkor a tizedespontot balra kell tolni. Például a 2.1E-5 0,000021-ot jelent. Gyakran nekünk is, kényelmes normál alakban megadni bizonyos számokat. A Delphi ugyanúgy értelmezi az alábbi három alakban megadott számot: 1000, 1E3 és 1E+3.

■ *Típuskonverziók*

Gyakran szükségünk van a sztringek számokká alakítására és fordítva. Ezt általában típuskonverziónak nevezik, mivel a változót egyik típusból a másikba alakítjuk. Például az IntToStr függvény segítségével Integer típusú változót alakíthatunk sztringgé. A FloatToStr vagy a FormatFloat függvény segítségével pedig egy valós típusú számot alakíthatunk karakterlánccá. Az alábbi példában az Üzenet egy sztring típusú változó:

```
Üzenet := 'A számláló pillanatnyi állása:' +
IntToStr(Számláló);
```

Megjegyzés: *Van néhány speciális feladatot megvalósító konverziós függvény (Val, Str, FloatToStrF és FormatStr). Rendelkezésünkre áll ezen kívül egy egész sor a dátum/idő sztringeket kezelő függvény (például: DateToStr, StrToDate, TimeToStr, FormatDateTime). Ezek közül a legfontosabbak valószínüleg a FloatToStrF és a FormatStr függvények. A FloatToStrF a FloatToStr függvény egy olyan bővített változata, amelyben meghatározhatjuk az eredmény sztring formátumát (l. beépített help).*

A hatodik fejezetben még lesz szó a típuskonverziókról.

5-3. táblázat. Alapvető konverziós függvények

Függvény	*Hatása*
IntToStr	Egészeket alakít sztringgé
IntToHex	Egészeket alakít hexadecimálissá
StrToInt	Sztringeket alakít egésszé. (A sztringnek az egész számot vagy annak hexdecimális alakját kell tartalmaznia. A StringToIntDef segítségével beállíthatunk egy alapértéket, amelyet akkor kapunk, ha a sztring nem megfelelő formátumú.)
FloatToStr	Lebegőpontos számból csinál karakterláncot.
StrToFloat	Egy lebegőpontos szám sztring-reprezentációját alakítja lebegőpontos számmá.

■ *Iterált műveletvégrehajtás-ciklusok*

Tegyük fel, hogy szükségünk van egy adott utasítás (lehet összetett is) többszöri megismétlésére. A programozásban egy programrész egymásutáni ismétlését – feladattól függően – különböző ciklusokkal valósíthatjuk meg. A ciklus belsejét ciklusmagnak nevezzük, itt található az az utasításblokk, amelyet iteráltan kívánunk végrehajtani. A Delphiben három különböző ciklusfajta áll rendelkezésünkre: az elöltesztelő, az elöltesztelő léptető és a hátultesztelő. Ezek közül az első és az utolsó hasonlít abban, hogy ezekben a ciklusmag végrehajtása addig ismétlődik, amíg valamilyen kezdeti feltétel nem teljesül. Ezeket *nem determinisztikus* ciklusoknak hívjuk, mert csak a cikluson belül dől el, hogy hányszor kerül végrehajtásra a ciklusmag. A léptető vagy *determinisztikus* ciklus rögzített számú alkalommal dolgozza fel a ciklusmagot.

Elöltesztelő léptető ciklus – for ciklus

Tegyük fel, hogy tíz egymást követő üzenetablakot kívánunk megjeleníteni egy eseménykezelőn belül. A legegyszerűbben ezt úgy érhetjük el, ha az eljárásban az alábbi programrészletet használjuk:

```
var
  I: Integer;
begin
  for I:= 1 to 10 do
    begin
      ShowMessage ('Ez az üzenetablak a' + IntToStr
      (I) + '-ik');
    end;
end;
```

A fenti utasítássorozatban a **for** és **to** fenntartott szavakat tartalmazó sor a következőt jelenti: „Az I vegyen fel minden egész értéket 1-től 10-ig, és minden értékadás után hajtsa végre:". A ciklust felfoghatjuk úgy is, mint egy felhúzott fogaskereket a számítógép belsejében, amely adott számú fordulatot tesz meg, és minden fordulat közben végrehajtja az általunk előírt feladatot. A feladatot a begin és end kulcsszavak közti utasítássorozat jelenti. (Ha csak egy utasítást teszünk a ciklusmagba – mint a fenti példában –, akkor nem szükséges a **begin/end** kulcsszópár, amely az összetett utasítás blokkját hivatott határolni, a gyakorlatban azonban a ciklusmagok legtöbbször összetettek.)

A **for** kulcsszó egy számláló-változót hoz létre. Az előzőekben ez a számláló egy egész típusú változó volt (I), amelynek 1-et adtunk kezdőértékül. A számlálás egyesével történik és jelen esetben 10-ig tart. Maga a folyamat így néz ki: a Delphi beállítja 1-re a számláló értékét, majd összehasonlítja a megadott végértékkel. Ha a végérték kisebb, akkor semmi nem történik. Amennyiben a a kezdőérték kisebb mint a végérték, akkor a Delphi végrehajtja a ciklusmagot, majd a záróutasítást követő pontosvesszőhöz érve eggyel növeli a számláló-változó értékét, és elöről kezdi a folyamatot. Az egész akkor ér véget, ha az aktuális számláló érték nagyobb, mint a kijelölt végérték. Ekkor a ciklus véget ér, és a Delphi a következő utasításra ugrik.

A for-ciklus szintaxisa általános esetben így néz ki:

```
for változó := kifejezés to kifejezés do
  begin {ciklusmag kezdete}
    utasítás1;
    utasítás2;
```

```
    .
    .
  end; {ciklusmag vége}
```

A számlálás nem mindig felfelé történik. Bizonyos feladatok a lefelé számlálást követelik meg. Ekkor a for-ciklusban a **downto** kulcsszót használjuk. Ez a kulcsszó jelzi a Delphinek, hogy a számlálót mindig eggyel csökkentse.

Például egy űrhajó-szimulációt megvalósító program nem lenne teljes az alábbi kódrészlet nélkül:

```
for I:= 10 downto 1 do
  begin
    ShowMessage (IntToStr (I)) ;
  end;
  ShowMessage ('Felszállás!') ;
```

A **downto** használatakor a ciklus végét az jelenti, ha a számláló értéke kisebb a kezdőértéknél.

Visual Basic Tipp: *A for-ciklus számlálója nem lehet valós szám, továbbá a Delphiben nincs a* ***step*** *kulcsszónak megfelelője. Ha egy ezek közül valamelyiket használó Visual Basic-beli for-next ciklust kell konvertálnunk, akkor a számlálót a cikluson belül kell növelnünk, illetőleg használjunk a ciklus belsejében más változókat, amelyek értékeit szintén a ciklusmagban változtatjuk.*

Egymásba ágyazott for-next ciklusok

Képzeljünk most el egy olyan táblázatot, amelynek egy adott sorában, egy adott kamatlábhoz tartozó kamatok jelennek meg különböző kiinduló összegekre kiszámolva. Egy rögzített oszlopon végigmenve pedig, az oszlophoz tartozó kiinduló összeg különböző kamatlábakhoz tartozó kamatait olvashatjuk le. Tegyük fel, hogy egy ilyen táblázatot a számítógépnek kell kitöltenie a kamatlábak és a megadott kiindulási összegek alapján. Kézenfekvőnek tűnik az az algoritmus, hogy végigmegyünk az első sor összes celláján, majd ezt megismételjük a második sorral stb., amíg el nem

jutunk az utolsó sorig. Ugyanilyen jó eljáráshoz vezet az, amikor nem a sorokon, hanem az oszlopokon megyünk végig. Ezeket az algoritmusokat a programban egymásba ágyazott ciklusok használatával valósíthatjuk meg. Példaként elkészítjük a szorzótáblát. Az alábbi programrészlet a 2-höz tartozó sort készíti el:

```
procedure TForm1.Button1Click(Sender: TObject);
var I : Byte;
begin
  for I := 1 to 10 do begin
    StringGrid1.Cells[0,I] := '2 x ' + IntToStr(I) +
    ' =' ;
    StringGrid1.Cells[1,I] := IntToStr(2 * I);
  end;
end;
```

Ahhoz, hogy az egész szorzótáblát megkapjuk, ezt a ciklust egy másikba kell ágyazni, amelyik a 2 helyébe 3 ír, majd a 3 helyébe 4-et és így tovább. A teljes eljárás így néz ki:

```
procedure Tform1.Button1Click(Sender: TObject);
var I, J : Byte;
begin
  for J := 1 to 10 do begin
    for I := 1 to 10 do begin
      StringGrid1.Cells[J,I] :=
      IntToStr(J) + 'x' + IntToStr(I) +
      ' = ' + IntToStr(J * I);
    end;
  end;
end;
```

Érdemes megfigyelni, mi történik: a J változó 1-ről indul, a Delphi belép a belső ciklusba, ahol az I számláló kezdeti értéke szintén 1. A Delphi 10-szer végrehajtja a belső ciklusmagot, majd a külső ciklus endjére ugrik. Ekkor megváltoztatja a J értékét 2-re, és kezdődik az egész folyamat elölről egészen addig, amíg a J értéke 10 nem lesz.

A for-ciklusok egymásba ágyazására vonatkozó szabály meglehetősen egyszerű: a belső ciklusnak véget kell érnie ahhoz, hogy a külső ciklus soron következő utasítása végrehajtható legyen. Lehetőség van háromszorosan, négyszeresen és akárhányszorosan egymásba ágyazott ciklusok létrehozására is. Határt csak ezek működésének áttekinthetősége szab, a Delphi nem.

Nem determinisztikus ciklusok

Programozás során gyakran van szükség nem determinisztikus ciklusokra, amelyekre az jellemző, hogy nem tudjuk előre meghatározni az iterációk számát. A Delphi két ilyen ciklusfajtával rendelkezik. Még mielőtt ezekre rátérnénk, fontos hogy tisztában legyünk az 5-4. táblázatban közölt relációkkal.

5-4. táblázat. Logikai relációk

Reláció	*Hatás*
=	Ellenőrzi, hogy fennáll-e az egyenlőség.
<	Ellenőrzi, hogy fennáll-e a *kisebb, mint* kapcsolat.
<=	Ellenőrzi, hogy fennáll-e a *kisebb vagy egyenlő, mint* kapcsolat.
>	Ellenőrzi, hogy fennáll-e a *nagyobb, mint* kapcsolat.
>=	Ellenőrzi, hogy fennál-e a *nagyobb vagy egyenlő, mint* kapcsolat.
<>	Megvizsgálja, hogy fennáll-e a *nem egyenlő* kapcsolat

Megjegyzés: *A Delphi alapértelmezésben nem értékeli ki teljesen a logikai kifejezéseket, ha nem szükséges. Magyarázatként nézzünk egy példát. Legyen a feltételünk A and B alakú (ahol A és B logikai kifejezések). Ha A hamis, akkor a feltétel mindenképpen hamis, tehát a B kiértékelése elmaradhat. Ezt a beállítást a Complete Boolean evaluation opción keresztül tehetjük meg a Project Options dialógusablak Compile oldalán. (A változtatás fölösleges, csak a program futását lassítja le!)*

A Repeat ciklus

A Repeat ciklusban a ciklusmag ismételten végrehajtódik, egészen addig, amíg a kilépési feltétel nem teljesül. Ez a fajta ciklus nem determinisztikus – azaz nem tudjuk előre, hogy hányszor fut le. Példaként számoljuk ki, hogy mennyi ideig kell 8 százalék kamatláb mellett 1000 Ft-ot bennhagyni a bankban, hogy a végén 1 000 000 Ft-unk legyen:

```
var
  Pénz: Double;     {A Delphi 32-ben használjuk a
                    Currency típust!}
  Count: Integer;
begin
  Money := 1000;
  repeat {ciklusmag kezdete}
    Pénz := Pénz * 1.08;
    Év := Év + 1;
  until Money > 1000000;  {tesztelés a ciklus végén}
ShowMessage(IntToStr(Év) + 'év szükséges.');
end;
```

Az általános sablon a Delphiben így néz ki ennél a ciklusfajtánál:

```
repeat
  begin
    Delphi utasítások;
  end;
until kilépési feltétel;
```

(A begin/end utasítás zárójel használata nem kötelező, mert a ciklusmagot elkülönítik a repeat/until fenntartott szavak.)

Megjegyzés: *Mivel a kilépési feltétel a ciklus végén értékelődik ki (itt dől el, hogy visszatérünk-e a ciklus elejére), ezért minden Repeat ciklus ciklusmagja egyszer legalább végrehajtódik. Az ilyen tulajdonságú ciklusokat a számítástechnikában hátultesztelő ciklusoknak nevezik.*

A logikai feltételnek nem kötelező egyszerűnek (pl. u=6) lennie. A kifejezésben felhasználhatjuk a logikai operátorokat (and-et, or-t, not-ot – még a xor-t is, ha úgy adódik) és természetesen a relációkat. Fontos viszont betartanunk, hogy a kapott kifejezés logikai legyen – valami, ami vagy igaz vagy hamis.

Visual Basic Tipp: *A Repeat ciklus a Delphiben ugyanaz, mint a Do-Until ciklus a Visual Basic-ben.*

A While ciklus

A While ciklus a Delphi harmadik típusú ciklusa. Ez a ciklus egy elöltesztelő ciklus, ami azt jelenti, hogy a kilépési feltétel a ciklus elején található. A szintaxisa a következő:

```
while feltétel do
  begin
    Delphi utasítások;
  end;
```

Ebben az esetben, mivel a kiértékelés felül van, az is elképzelhető, hogy a ciklusmag egyszer sem hajtódik végre. Itt szükséges a begin/end használata is, továbbá (megint a kiértékelés helye miatt) fontos ellenőriznünk, hogy van-e egyáltalán értelme a feltételnek. A feltétel teljesülésével a ciklusmag végrehajtódik, ezért ott utólagos változtatásokra már nincs módunk – nem úgy, mint a Repeat ciklusban.

■ *Feltételes utasításvégrehajtás*

Most már képesek vagyunk olyan programokat írni, amelyek eldöntik, hogy megismételjenek-e egy adott utasítássorozatot. Olyat még nem, amelyik eldönti – bizonyos feltételtől függően –, hogy melyik utasítást hajtsa végre. A következő néhány oldalon ezzel is megismerkedhetünk. Ebben a fejezetrészben ismertetésre szánt összes utasítás azt az egyszerű

dolgot próbálja megvalósítani programozás szinten, ami valahogy így hangzik:

Ha valami teljesül, akkor csináld ezt, különben csináld azt.

A Delphiben (és sok más programnyelvben) pontosan ezt a ha-akkor szerkezetet használjuk, csak éppen angolul. Az **if** a *ha*-nak felel meg a **then** pedig az *akkor*-nak. Például, ha figyelmeztetni akarjuk a felhasználót, hogy valamilyen általa megadott értéknek pozitívnak kell lennie, akkor használhatjuk az alábbi programsort:

```
if. I < 0 then
  ShowMessage ('A számnak pozitívnak kell lennie!')
```

Még általánosabban: ha a Delphi egy if-then utasításpárt talál, akkor először ellenőrzi az első tagmondatot (az if-es részt) – amely egy logikai állítás (a logikai műveletek és relációk felhasználásával) –, és ellenőrzi igazságtartalmát. Ha igaz, akkor a számítógép a then utáni utasítást (utasítássorozatot) hajtja végre. Ha a feltétel hamis, akkor a végrehajtás a következő utasításra ugrik.

Gyakran két programágra is szükségünk lehet, ekkor az if-then utasításpár kevés, hiszen itt elágazás csak akkor történik, ha a feltétel igaz. Ha azt szeretnénk, hogy a feltétel hamis voltakor is történjen valami (végrehajtódjon egy utasítássorozat), akkor az if-then (ha-akkor) mellett az else (különben) utasítást is használnunk kell. Ekkor mindenképpen szükséges a begin/end használata az utasítások elhatárolására. Nézzük meg, hogy néz ez ki általánosan:

```
if valami igaz then
  begin
    utasítások;
  end
else
  begin
    utasítások;
  end;
```

Megjegyzés: Ha az if-then-else utasításhármast használjuk, akkor nem teszünk kettőspontot az if blokkot lezáró ***end*** *után, sem az* ***else*** *kulcsszó után, amely az* ***else*** *blokk utasításait vezeti be.*

Természetesen az else blokk opcionális; használatával azt jelezzük, hogy egy utasítás vagy utasítássorozat következik. Ez a blokk csak akkor hajtódik végre, ha az **if** utáni feltétel hamis.

Példaként nézzük meg az előző példa egy módosított változatát:

```
if I < 0 then
  begin
    ShowMessage ('A számnak pozitívnak kell lennie!');
  end
else
  begin
    ShowMessage ('A szám pozitív volt!');
  end;
```

Most is, mint mindig, a bekezdések a program olvashatóságát szolgálják, a fordítót a „külalak" egyáltalán nem érdekli.

If-then utasítások használata ciklusokkal

Ciklusok használatakor gyakran felmerül az igény, hogy bizonyos feltétel teljesülése esetén idő előtt kiszálljunk vagy visszakerüljünk a ciklus elejére, a törzs hátralevő részét átugorva, azaz új iterációval folytatódik a ciklus. A **continue** kulcsszó éppen ez utóbbit valósítja meg. Például, ha babonásak vagyunk:

```
var
  Számláló: Integer;
begin
  for Számláló :=1 to 100 do
    begin
      if. Számláló = 13 then
        continue;
```

```
      ShowMessage (IntToStr(Számláló));
    end;
end;
```

A program nem irja ki a 13-at, mert ha a számláló értéke 13, akkor a **continue** kulcsszó hatására a végrehajtás a ciklus végére ugrik az üzenetablak megjelenítése előtt.

A **continue**-hoz hasonlóan működik a **break** kulcsszó is, hatására a program futása a ciklus utáni első utasítással folyatódik. Példaként az előző program módosított változatát mutatjuk be, amely csak az 1 és 12 közötti számokat jeleníti meg.

```
var
  Számláló: Integer;
  begin
    for Számláló :=1 to 100 do
      begin
        if. Számláló = 13 then
        break;
        ShowMessage (IntToStr(Számláló));
      end;
  end;
```

Tipp: A megfelelő if-then utasításpárral használt ***break*** *kulcsszóval elérhetjük, hogy a ciklus középen-tesztelő legyen.*

A case utasítás

Tegyük fel, hogy egy olyan programot szeretnénk készíteni, amely négy vizsga eredményének átlagát alapul véve, kiszámolja az érdemjegyet. A jeles osztályzat 90 pont fölötti átlagnak felel meg, jó-t az kap, aki 80 és 89 pont közti átlagot produkált és így tovább. Ez egy gyakori programozási szituáció, amelyet a Delphiben a **case** szerkezettel kezelhetünk. A **case** szerkezet jelzi, hogy a programunk egy többszörös elágazási ponthoz ju-

tott. (A több if-then szerkezet egymásutáni használata bonyolult és kevésbé áttekinthető, ami a hibakeresést is megnehezíti.)

Például az alábbi kódrészlettel értékelhetjük ki a PontÁtlag változót.

```
case Pontátlag of
     100:        ShowMessage ('jeles+!');
     90 . . 99:  ShowMessage ('jeles');
     80 . . 89:  ShowMessage ('jó');
end;
```

A **case** kulcsszót követő változót vagy kifejezést *szelektornak* nevezzük. A szelektor típusa Char, vagy valamilyen egész kell hogy legyen. Az, hogy a program a **case** szerkezet melyik ágán folytatódik, kizárólag a szelektorváltozó értékétől függ. Az ágakat jelentő utasításokat kettőspont előzi meg. (Itt is lehetőségünk van összetett utasítások használatára, ha az adott ágon egyszerre több dolgot akarunk megvalósítani.)

Az egyes ágak egy konstanssal, (egymástól vesszővel elválasztott) konstansok sorozatával vagy egy konstansok által határolt egész intervallummal (két pontot teszünk az intervallum határait jelölő két konstans közé) van megcímkézve, ezeket hívjuk **case** állandóknak. A **case** szerkezetet kibővíthetjük az else kulcsszóval, amivel összefoghatjuk a nem vizsgált eseteket. Itt is igaz az általános szabály, amely szerint az egyes utasításokat a begin és az end kulcsszavak közé írt összetett utasítással helyettesíthetjük.

Megjegyzés: *Csak egy* ***case*** *ág hajtható végre, ezért nem használhatunk egymást fedő feltételeket a* ***case*** *szerkezetben.*

A következő példa az angol betűkről dönti el, hogy azok magánhangzók-e vagy mássalhangzók:

```
case UpCase (chA) of
       'Y':.  begin
                ShowMessage ('Néha magánhangzó');
                ShowMessage ('Nehéz ügy');
              end
```

```
        'A', 'E', 'I', 'O', 'U' :
                ShowMessage ('Magánhangzó');
else
        ShowMessage ('Mássalhangzó');
end;
```

■ *Objektumok kezelése futási időben*

Vannak olyan objektumok és objektumtulajdonságok, amelyekhez csak a programon keresztül lehet hozzáférni – sem a tulajdonságok sem pedig maguk az objektumok nem elérhetőek az objektumfelügyelőn keresztül. Például ha azt szeretnénk megtudni, hogy hány soros a képernyő, akkor a TScreen objektum Height tulajdonságának értékét kell kiolvasnunk, de a TScreen és az ehhez kapcsolódó tulajdonságok csak futási időben érhetők el. Hasonlóképpen a nyomtató vezérlését kizárólag futási időben végezhetjük a TPrinter objektumon keresztül.

Megjegyzés: *Majdnem minden Delphi objektumról létrehozhatunk példányokat (instance) futási időben. A 7. fejezetben megmutatjuk hogyan.*

Metódusok

Nyilvánvalóan szeretnénk, hogy objektumaink csináljanak is valamit, ekkor leggyakrabban a Delphi beépített metódusaihoz fordulunk.. Például, ha a TEdit komponens tartalmát kívánjuk törölni, akkor az alábbi sor használatával érhetünk célt:

```
Edit1.Clear;
```

Mivel a Delphi többszáz metódust tartalmaz (csak a TEdit komponenshez 32 kapcsolódik), lehetetlen feladat lenne mindegyiket ismertetni. Forduljunk a beépített help-hez, ha meg akarjuk tudni, hogy egy adott komponens rendelkezik-e a kívánt célt megvalósító metódussal. Úgy mint a tulajdonságok esetében, a metódusok közti eligazodást is

könnyebbé teheti az a tény, hogy ugyanolyan nevű metódusok hasonlóképpen viselkednek. A Clear metódust például majdnem minden szöveget kezelő vezérlőelemre alkalmazhatjuk, hatása minden esetben a szöveg törlésében nyilvánul meg. Egy másik példa a SetFocus metódus, amely minden olyan komponensre alkalmazható, amely megkaphatja a fókuszt, hatására – ahogy az angol elnevezés sugallja – az adott komponensre kerül a fókusz.

Megjegyzés: A Delphi metódus hatását egy komponensre úgy változtathatjuk meg, hogy létrehozzuk a komponens mását, mint saját készítésű komponenst. A módszer pontos leírása megtalálható a Delphihez mellékelt „Component Writers Guide" című dokumentációban.

A következőkben a leggyakrabban használt metódusok közül ismertetjük a fontosabbakat. A későbbi fejezetek során megismerkedhetünk más speciális metódusokkal is. (Például a következő fejezetben szó lesz a TListBox és a TComboBox komponens leggyakrabban használt metódusairól. A 12. fejezetben pedig a grafikával kapcsolatos metódusokat ismerhetjük meg.)

BringToFront, SendToBack. Ez a metódus a komponenst (vagy formot, ha több formmal dolgozunk) a többi komponens vagy form fölé (vagy mögé) helyezi (L. a Z-Order-ről szóló alfejezetet a 13. fejezetben.)

Create. Ennek a fontos metódusnak a használatával hozhatunk létre új példányokat Delphibeli objektumokról futási időben. (Részletesen a 8. fejezet tárgyalja.)

Focused. Ez példa azokra a metódusokra, amelyek viselkedése hasonlít a tulajdonságokéra. Ennek a logikai metódusnak a segítségével megállapíthatjuk, hogy a fókusz az adott vezérlőelemen van-e. Általában az alábbihoz hasonló kontextusban használjuk:

```
if Edit1.Focused then ...
```

Hide, Show. A Hide a formot vagy vezérlőelemet láthatatlanná teszi. Ugyanaz, mint a Visible tulajdonság False-ra állítása. A Show metódussal

ismét láthatóvá tehetjük az elrejtett formot vagy komponenst. A Show hatására a BringToFront metódus is végrehajtódik egy alul lévő form esetében (l. „Több formot használó alkalmazások" részt).

Refresh, Repaint. A Refresh metódus letörli majd újrarajzolja a komponenst. A Repaint csak az újrarajzolást végzi el, de nem töröl semmit a képernyőről (l. még a 12. fejezetet).

ScaleBy. Segítségével arányosan nagyíthatunk ill. kicsinyíthetünk komponenseket. Például a Ccommand1 nevű TButton komponens 50 százalékkal történő nagyítását valósítja meg a

```
Command1.ScaleBy(3, 2);
```

sor. Az első szám a méretarány számlálójaként, a második pedig a nevezőjeként fogható fel. A példából leolvasható arány 3/2 (150 százalék).

Tipp: Ha a felhasználó megváltoztatja a form méretét, szükségünk lehet a komponensek áthelyezésére és újraméretezésére. Használjuk ehhez a form (vagy a tartalmazó komponens) különböző mérettulajdonságait.

A TApplication objektum

Minden Delphi projekt futtatásakor automatikusan létrejön a TApplication osztály egy példányaként az Application objektum. Ennek az objektumnak a tulajdonságaival és metódusaival tudjuk megadni az alkalmazás ikonját, vagy a projekt help fájljának nevét (l. 12. fejezet). Az alábbiakban röviden ismertetjük a TApplication objektum legfontosabb tulajdonságait és metódusait. (Amennyiben a TApplication objektumhoz kívánunk eseménykezelőt írni, olvassuk el a beépített súgó ide vonatkozó részét. Az itt alkalmazandó technikák különböznek azoktól, amelyeket más fejlesztési időben elérhető objektumok eseménykezelőinek írásánál használunk.)

Terminate metódus. Ennek a metódusnak a meghívása (Application.Terminate) a szabványos módon vet véget az alkalmazás futásának.

Hatása megegyezik a Halt eljárásával, de a programozók zöme ezt választja, mert elképzelhető, hogy a Halt eljárást nem fogják támogatni a Delphi jövőben megjelenő változatai.

ExeName tulajdonság. Ez a kizárólag olvasható tulajdonság adja meg a lefordított projekt teljes elérési útját, az .EXE kiterjesztés elhagyásával. Például a C:\DELPHI\PROJECTS\PROJECT1.EXE nevű fájl esetén a tulajdonságra az

```
Application.ExeName =
'C:\DELPHI\PROJECTS | PROJECT1'
```

állna.

HelpFile tulajdonság. Itt adható meg a Help fájl neve és elérési útja (l. 13. fejezet).

Icon tulajdonság. Az alkalmazás minimalizálásakor az ebben a tulajdonságban megadott ikon lesz látható. Tegyük fel, hogy van egy pillangót ábrázoló BTFLY.ICO ikonunk a C meghajtó Icons könyvtárában. Ezt így rendelhetjük a projekthez:

```
Application.Icon.LoadFromFile ('C:\ICONS\BTFLY.ICO') ;
```

Minimize, Restore metódusok. A Minimize metódus hatására alkalmazásunk minimalizálódik (jelenlétét az előző tulajdonságban beállított ikon jelzi). Ez különösen több formból álló projektek esetén hasznos. Amikor a felhasználó minimalizálja az alkalmazást, akkor is ez a metódus hívódik meg. A Restore segítségével visszaállíthatjuk a minimalizált állapotból az azt megelőző állapotot.

ProcessMessages metódus. Ez egy rendkívül fontos metódus. Használatával az eseménysor folyamatos olvasását biztosíthatjuk a Windows számára. A Windows 3.X -ben a multi-tasking (látszólagos párhuzamos futtatás) működéséhez az kell, hogy az alkalmazások együttműködjenek az operációs rendszerrel a processzoridő kiosztásában, ezért ezt a metódust használjuk olyan eseménykezelőkben, amelyek sokáig lefoglalják a

processzort. (Windows 95 használata esetén is hasznos lehet, amikor régebbi Windows verzió alá írt programot futtatunk, és akkor is, ha idő előtt vissza akarjuk adni a processzorhasználat jogát az operációs rendszernek.)

A ProcessMessage metódus néhány gyakori alkalmazása:

- Használható olyan alkalmazásban, amely időigényes számításokat végez
- Használható olyan Delphi alkalmazásokban, amelyek együttműködnek más Windows programokkal (ilyen például egy terminálprogram, amely más alkalmazástól fogad adatokat).

Megjegyzés: A Delphi 32-ben több thread-et adhatunk ugyanahhoz a. programhoz. (L. még a beépített help-et.)

Title. Az itt beállított szöveg jelenik meg az ikon alatt.

A TScreen objektum

Ez az alapvető objektum ad felvilágosítást a Windows képernyőtulajdonságairól. Például a Screen.Height értéke megmondja, hogy a Windows milyen sorfelbontást használ (pixelekben). Ebben a szakaszban a TScreen objektum leggyakrabban használt tulajdonságait mutatjuk be.

Az ActiveControl és az ActiveForm tulajdonság. Ezek határozzák meg, hogy melyik vezérlőelemen ill. formon van a fókusz. Az ActiveForm akkor létfontosságú, ha olyan több formot használó alkalmazással dolgozunk, amelyben előírjuk, hogy a felhasználó melyik formmal dolgozhat.

ComponentCount, ComponentIndex, Component, Forms, FormsCounts tulajdonságok. Ezeket akkor használjuk, ha valamilyen keresés során végig kell néznünk egy alkalmazás formjait vagy az egy formhoz tartozó komponenseket. (L. még a 7. fejezetet.)

Cursor tulajdonság. Ellentétben a komponensek Cursor tulajdonságával, a Screen objektum Cursor tulajdonságának globális hatása van. Ezzel az

alkalmazás összes formjában megjelenő kurzor alakját állíthatjuk be. (Ezt az egyes komponensekben azonban felülírhatjuk a vezérlőelem saját Cursor tulajdonságának segítségével.)

Height, Width tulajdonságok. Ez a képernyő függőleges és vízszintes irányú felbontását adja meg pixelben a Windows jelentése alapján.

PixelsPerInch tulajdonság. Ez a képernyő-meghajtó által jelenleg használt felbontást adja meg pixel/inch mértékegységben.

Fonts tulajdonság. A Fonts tulajdonság az elérhető képernyőfontokat tartalmazza. Ez a tulajdonság egy sztringlistát ad meg. (A sztringlistákat a könyv a 6. fejezetben tárgyalja.)

Cursors tulajdonság. Ez a tulajdonság az elérhető kurzorokat adja meg, szintén sztringlista formájában.

A TCanvas objektum

A TCanvas objektum egy másik példája azoknak a tulajdonságoknak, amelyek maguk is objektumok. Miként azt a neve sugallja, a TCanvas egy olyan objektum, amely egy rajzolási felületet (vásznat) biztosít. Használatával szöveg és grafika is megjeleníthető. Ebben a szakaszban megtanulhatjuk, hogyan helyezhetünk el segítségével szöveget a form kívánt részében. A következő alfejezetben láthatjuk majd, hogy használatával hogyan pozicionálhatjuk a szöveget nyomtatáskor. (A 12. fejezetben tárgyaljuk, hogy a TCanvas hogyan használható grafikák készítéséhez a formon vagy nyomtatón).

A szöveg megjelenítésében kulcsszerepet a TextOut nevű metódus játssza. Használatát az alábbi példa mutatja be:

```
Form1.Canvas.TextOut ( 0, 0, 'Ez most a képernyő
bal felső sarkában jelenik meg');
```

Az általános szintaxis így néz ki:

Objektumnév.Canvas.TextOut(*Vízszintes koordináta, Függőleges koordináta, Szöveg*)

ahol a koordináták a megjelenítendő szöveg bal felső sarkának helyét határozzák meg, a szöveg egy sztring vagy sztring típusú kifejezés, amely a megjelenítendő szöveget jelöli ki.

A szöveget gyakran csak méreteinek (magasság, szélesség) ismeretében tudjuk esztétikusan elhelyezni az adott komponensre. A szöveg méreteit a TextHeight (magasság) és a TextWidth (szélesség) metódusokkal olvashatjuk ki. A szintaxis mindkét esetben hasonló. Például a

```
TempTextWidth := Form1.Canvas.TextWidth
('Ez itt a szöveg helye');
```

sorral a TempTextWidth változóban megkapjuk, hogy hány pixelt foglal el vízszintesen az „Ez itt a szöveg helye" felirat. Ezen információ birtokában alkalmasan átméretezhetjük komponenseinket és formjainkat.

A TPrinter objektum

A Delphi mindent elkövet annak érdekében, hogy elkerüljük azt a zűrzavart, amelyet a különböző nyomtatók kezelése okozhat. A. TPrinter objektum tulajdonképpen a Windows printer-meghajtójához csatlakozó nyomtatási felület. Így nyomtatáskor csak a TPrinter objektum tulajdonságaira és metódusaira van szükségünk.

A TPrinter használatakor nem szabad megfeledkeznünk egy fontos szabályról: a unit **uses** blokkjában felsoroltakhoz vegyük hozzá a **printers** kulcsszót is. A **uses** blokk a Delphi 16-ban valahogy így néz ki:

```
uses
  SysUtils, WinTypes, WinProcs, Classes, . . .
```

A Delphi 32-ben pedig így:

```
uses
  Windows, Messages, SysUtils, . . .
```

Keressük meg a kódszerkesztőben ezt a sort, és illesszük a felsoroltak közé bárhová (de még a záró pontosvessző elé) a **printers** kulcsszót. Így tehetjük elérhetővé a Printers unitot projektünk adott modulja számára.

Tipp: *Egy adott form kinyomtatásához használjuk a form Print metódusát.*

Ha a formokon kívül más dokumentumokat is ki szeretnénk nyomtatni, akkor szükséges inicializálni a printert. Amikor az adatok küldése a nyomtatóhoz véget ért, le kell zárnunk a kapcsolatot. Mindig tartsuk be a következőket:

- A nyomtatás elkezdéséhez használjuk a BeginDoc metódust.
- A nyomtatás végeztével hívjuk meg az EndDoc metódust.

(Fel kell készülnünk bizonyos nyomtatási hibák – pl. kifogyott a papír – fogadására is. Ehhez azonban tisztában kell lennünk a kivételek kezelésével, amivel majd a 8. fejezetben foglalkozunk.)

Rendszerint a printer Canvas tulajdonságának (objektumának) használatával továbbíthatjuk a grafikákat vagy szöveges adatokat a nyomtatóhoz. A szintaxis hasonló a formok Canvas objektumainál látottakhoz. Például:

```
Printer.Canvas.TextOut (0, 0, 'A bal felső sarokban
megjelenő szöveg helye');
```

Megjegyzés: *A printer Canvas objektumának legtöbb tulajdonsága (pl. TextOut, TextHeight és Fonts) hasonlóképpen működik formokra történő nyomtatás esetén, mint a printer használatakor.*

Általánosan igaz, hogy a Canvas tulajdonság X, Y koordinátája segítségével megadhatjuk az egyes pontok helyét a nyomtatandó oldalon.

Megjegyzés: *Egy memóablak tartalmának a nyomtatásához hasonló technikákat alkalmazunk, mint a fájlkezelésnél.*

Az alábbiakban a legfontosabb printer-tulajdonságokat és metódusokat követhetjük végig.

Abort metódus. Segítségével megszakíthatjuk az aktuális nyomtatási munkát, például ha egy hiba történik.

Printing tulajdonság. Ez a logikai tulajdonság felvilágosítást ad arról, hogy a kívánt nyomtatás folyamatban van-e még. (Természetesen, ha a printer nagy pufferral rendelkezik, akkor ez elvesztheti jelentőségét, hiszen lehet hogy a szöveg a pufferban van, és ilyenkor a tulajdonság megbízhatatlanná válik.)

Printers, PrintIndex tulajdonságok. A printers tulajdonság egy sztringlistát tartalmaz (l. 6. fejezet), amelyben az adott számítógépen elérhető nyomtató meghajtók vannak felsorolva. A PrinterIndex tulajdonság értéke az éppen kijelölt printer neve.

NewPage metódus. Ez a metódus a lapdobást kiváltó vezérlőkódot küldi el a printernek.

Orientation tulajdonság. Ez mondja meg, hogy az aktuális nyomtatási kép milyen állású (álló vagy fekvő).

PageHeight, PageWidth tulajdonságok. Az aktuális oldal magasságát és szélességét adja meg pixel mértékegységben. Például egy 300 dpi felbontású lézernyomtató és egy 8,5×11 inch-es lap esetén a PageHeight értéke 3150, míg a PageWidth-é 2400 lesz, ami egy 8×10,5 inch-es nyomtatási felületnek felel meg.

PageNumber tulajdonság. Ez az aktuális oldalszámot adja vissza.

PrintScale tulajdonság. (Ez szigorúan véve inkább a form tulajdonsága, mintsem a TPrint objektumé.) Ha egy formot a Print metódus használatával nyomtatunk, ebben a tulajdonságban határozhatjuk meg, hogy a formot változatlan méretben nyomtatjuk-e, vagy esetleg arányos méretváltoztatást akarunk végrehajtani. A tulajdonság három különböző értéket vehet fel: a poNone-t, a poPropotional-t (amely megtartja a képer-

nyőn használt arányokat) és a poPrintToFit-et (amelyik szintén megtartja a képernyő arányait, de biztosítja azt is, hogy a form ráférjen az oldalra).

■ *Üzenetablakok és inputablakok*

Speciális célú üzenetablakok használata meglehetősen elterjedt a Windows alkalmazások körében. Találkoztunk már az alapokkal: a ShowMessage eljárás segítségével meg tudunk jeleníteni üzenetablakokat. A használt szintaxis ShowMessage(*sztring*) alakú.

Sajnos a ShowMessage eljárás hatása kimerül egy a megadott sztringet tartalmazó ablak és egy OK feliratú nyomógomb megjelenésében. (Az üzenetablak fejlécében pedig az alkalmazást tartalmazó végrehajtható fájl neve szerepel.)

Megjegyzés: *A ShowMessagePos eljárás segítségével az üzenetablakot egy általunk megjelölt pozícióban jeleníthetjük meg.*

Az üzenetablak felületén a felhasználóval nemcsak közölhetünk bizonyos dolgokat, hanem van mód választási lehetőségek eljuttatására is, ezt különböző nyomógombok beillesztésével tehetjük meg.

Megjegyzés: *Minden üzenetablak modális. Ez azt jelenti, hogy ezeket be kell zárni, mielőtt az alkalmazás tovább folytatódna.*

Két üzenetablakot megjelenítő függvény létezik a Delphiben:

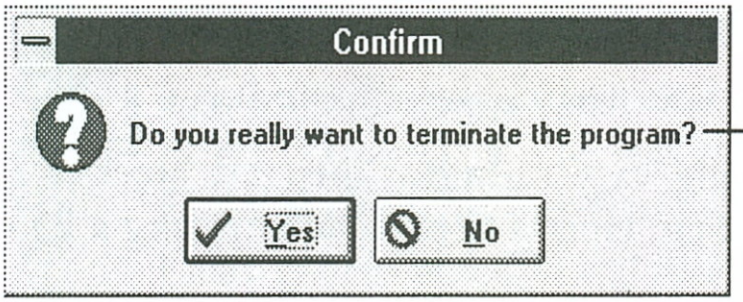
- Ez a függvény egy szöveges, fejléccel és megadott nyomógombokkal rendelkező üzenetablakot jelenít meg.
- A MessageDlgPos függvény, amellyel a megjelenítendő üzenetablak helyét is megadhatjuk.

A MessageDlg függvény

Mielőtt megvizsgálnánk az üzenetablak általános szintaxisát, nézzük meg mit csinál az alábbi töredék:

```
if. MessageDlg ('Tényleg ki akar lépni
a programból?', mtConfirmation, [mbYes, mbNo], 0) =
mrYes then Application.Terminate;
```

Az elvárásoknak megfelelően az alábbi üzenetablak jelenik meg a képernyőn:



Az üzenetablak egy Yes és egy No gombbal rendelkezik. Általában igaz, hogy amikor a felhasználó rákattint valamelyik gombra, a Delphi a lenyomott gomb típusától függően egy értéket ad vissza. A példánknál maradva: ha a felhasználó a Yes gombra kattint, a Delphi az előredefiniált mrYes konstanst adja vissza, ha a No gombot nyomja le, akkor a függvény az mrNo értékkel tér vissza. A fenti sor megvizsgálja, hogy a függvény értéke egyenlő-e az mrYes konstanssal. Ha igen, akkor akkor az alkalmazás leáll.

A MessageDlg függvény hívásának általános szintaxisánál a beépített súgóban ezt találjuk:

```
MessageDlg (const Msg: string; AType: TMsgDlgType;
AButtons: TMsgDlgButtons; HelpCtx: Longint): Word;
```

A beépített súgó egészére ez a jelölésrendszer jellemző, ezért érdemes egy kevés időt ennek tételes vizsgálatára szánni. A függvényeknek átadott információkat az ún. *paraméterek* hordozzák a Delphiben. (A hatodik fejezetben bővebben foglalkozunk majd a különböző paraméterekkel.) A paraméter típusa, csakúgy mint a deklarációkban, a kettőspont mögött van feltüntetve. A különböző paramétereket pontosvesszők választják el egymástól. A paraméter értékét a függvény csak akkor változtathatja meg, ha azt a **var** kulcsszó előzi meg, ha előtte a **konst** fenntartott szó áll, akkor nem.

Figyeljük meg, hogy a függvény paramétereinek típusai között sok ismeretlen található. A könyvtári függvényeknél előforduló ezen típusok általában konstansok valamilyen halmazának felelnek meg. Pédául a TMsgDlgType az üzenetablak fajtáját leíró konstansokból áll. Ezeket röviden összefoglaljuk.

Megjegyzés: *A felsorolt típusokkal, amelyek bizonyos dolgok listájaként tekinthetők, bővebben a 6. fejezet foglalkozik.*

Az Msg parameter. Ez egy sztring vagy szöveges kifejezés, amelyet a Delphi megjelenít az üzenetablakban. Hossza nem haladhatja meg a 255 karaktert.

Az AType paraméter. Ez a paraméter az üzenetablak típusáért felelős. A típusokat az 5-5. táblázat rendszerezi. A különböző üzenetablakok megjelenítéséhez különböző étékeket kell adnunk a TMsgDlgType típusú paraméternek.

5-5. táblázat. Üzenetablakok fajtái

A TMsgDlg paraméter értéke	*Az üzenetablak fajtája*
mtWarning	Sárga felkiáltójelet használ.
mtError	Piros stoptáblát használ.
mtInormation	Kék *i* betüt használ.
mtConfirmation	Zöld kérdőjelet használ.
mtCustom	Nem engedélyezi a bittérkép használatát, és a fejlécben az alkalmazás .EXE fájljának a nevét szerepelteti.

Az AButtons paraméter. Ez a paraméter határozza meg, hogy az üzenetablakban milyen nyomógombok jelenjenek meg. Több gomb hozzáadása úgy lehetséges, hogy a gombokat szögletes zárójelben soroljuk fel, egymástól vesszővel elválasztva, mint ahogy azt a példában is tettük a Yes és No gombok beillesztésekor: [mbYes, mbNo]. (Az AButtons halmaz típusú paraméter – l. 6. fejezet.) Az 5-6. táblázat a különböző gombfajtákat mutatja be.

A példában már láttuk, hogy a MessageDlg függvény visszatérési értékének vizsgálatával határozhatjuk meg, hogy melyik gombot nyomta le a felhasználó. A gombok értékeit hordozó előredefiniált konstansok nevükből adódóan könnyen átláthatóak. Az mrCancel például a Cancel gomb lenyomását jelzi, mrNo a No gomb lenyomásakor visszaadott értékkel rendelkezik és így tovább. (A teljes listát megtaláljuk a beépített súgóban.)

Megjegyzés: Három előre definiált konstans áll rendelkezésünkre, a gyakran előforduló gombsorozatok jelölésére. Ezek az mbYesNoCancel (a Yes, No, Cancel gombhármas jelölésére), az mbOkCancel (az Ok, Cancel gombpár jelölésére) és az mbAbortRetryCancel (az Abort, Retry, Cancel gombhármas jelölésére) nevű konstansok. Ezeket természetesen nem tesszük szögletes zárójelek közé.

A HelpCtx paraméter. Ez a paraméter tartalmazza a megfelelő csatlakozási pontot a program help fájljához. (L. 13. fejezet.)

A MessageDlgPos függvény

A MessageDlg függvény a képernyő közepére helyezi az üzenetablakot. A MessageDlgPos függvényt kell használnunk akkor, ha az üzenetablakot egy általunk meghatározott pozícióban kívánjuk megjeleníteni.

5-6. táblázat. Az üzenetablakhoz választható nyomógombfajták

Érték	*Ez látható a felületén*
mbYes	Egy zöld pipa jel és a Yes felirat.
mbNo	Egy piros áthúzott kör és a No felirat.
mbOK	Egy zöld pipa jel és az OK felirat.
mbCancel	Egy piros X és a Cancel felirat.
mbHelp	A Help felirat.
mbAbort	Egy piros pipa jel és az Abort felirat.
mbRetry	Két görbe nyíl egy kör mentén és a Retry felirat.
mbIgnore	Egy lépő zöld ember és az Ignore felirat.
mbAll	Egy zöld kettős pipa jel és az All felirat.

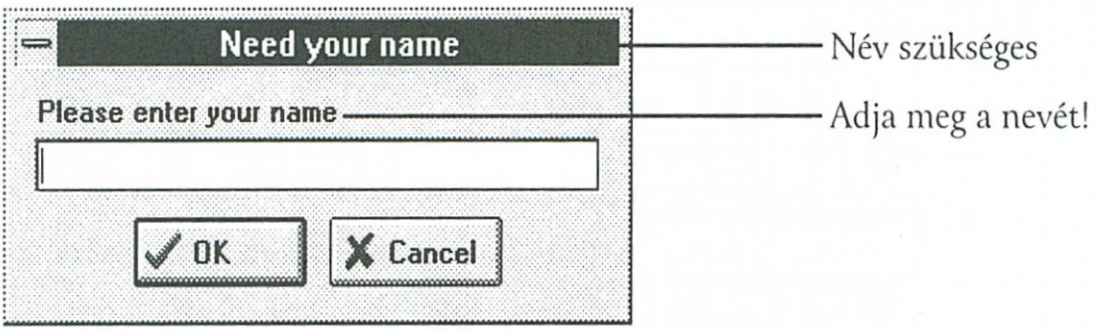
A szintaxis nagyon hasonló a MessageDlg függvényéhez, leszámítva a két paramétert, amelyek a kívánt pozíció koordinátáit hordozzák (pixel mértékegységben).

```
function MessageDlgPos (const Msg: String; AType:
TMsgDlgType; AButtons: TMsgDlgButtons; HelpCtx:
Longint; X, Y: Integer): Word;
```

Az X és Y egész paraméterek azokat a (pixelben megadott) képernyőkoordinátákat tartalmazzák, ahová az üzenetablak bal felső sarkát kívánjuk elhelyezni.

Az InputBox függvény

A TEdit és a TMemo komponensek a leggyakrabban használt eszközök a felhasználótól várt adatok bekérésére. Ennek ellenére néha szükség lehet az alkalmazás megállítására egy modális dialógusablak megnyitásával, amelyben a felhasználót valamilyen lényeges (és általában rövid) információ megadására kérhetjük. Bizonyos esetekben jelszó megadására kell kötelezünk a felhasználót, mielőtt az alkalmazás törzse lefuthatna. Ekkor az InputBox függvényt használjuk, amely egy egysoros szöveg bekérését teszi lehetővé. A függvény hatására az itt látható dialógusablak jelenik meg:



A következő programsor edményezte a fenti inputablak megjelenését:

```
FelhasználóNév := InputBox('Név szükséges',
'Adja meg a nevét' , '');
```

Az InputBox függvény általános szintaxisa így néz ki:

SztringVáltozó:= InputBox(const: *ACaption, APrompt, ADefault:* string);

(Az adatbeviteli ablakok függvények, azaz rendelkeznek valamilyen visszatérési értékkel. A 6. fejezetben többet megtudhatunk még a függvényekről.)

Az ACaption paraméterben az ablak fejlécében megjeleníteni kívánt szöveget adhatjuk meg, míg az APromt paraméter az adatbeviteli ablakban megjelenő felszólítás vagy kérdés szövegét tartalmazza. Végül az ADefault paraméternek egy olyan sztringet adunk értékül, amelyet az adatbeviteli ablak megjelenésének első alkalmával szeretnénk kiíratni a szerkesztési sorban. Az üres idézőjelpár használatával ez a funkció kikerülhető. Ha a felhasználó a Cancel gombot nyomja le (ugyanez érhető el az ESC leütésével is), akkor az InputBox függvény az alapértelmezésbeli értéket adja vissza. Ha a felhasználó az OK gombot nyomja le, akkor a szerkesztési sorba beírt sztring lesz a függvény visszatérési értéke.

Az InputQuery függvény

Az InputBox függvény egy általánosabb változata, amelyben azt is megállapíthatjuk, hogy a felhasználó, melyik gombot nyomta le (az Ok-t vagy a Cancel-t). A szintaxis szinte alig különbözik az InputBox-étól:

function InputQuery (const *ACaption, APrompt*: string;
var *Value*: string);

Az ACaption és az APrompt paraméterek az előzőekben leírtaknak megfelelően viselkednek. Itt azonban a Value paraméter az a sztring, amely az ablak első megnyílásakor a szerkesztési sorban szerepel majd. A **var** kulcsszó miatt ez a paraméter megváltozik és a felhasználó által a szerkesztési területen megadott sztringgel lesz egyenlő, ha a felhasználó a gépelés után lenyomja az ENTER-t. Emellett a függvény visszatéréskor egy logikai értéket ad vissza, amelynek igaz vagy hamis volta attól függ, hogy a felhasználó az OK vagy a Cancel gombot nyomta-e le. Vegyük észre, hogy szükségszerű volt egy változtatható paraméter bevezetése, hiszen egyébként a függvény nem tudna két értéket visszaadni.

Nézzünk most egy példát, amelyik illusztálja az InputQuery függvény használatát. Kezdjünk egy új projektet, és adjuk hozzá az alábbi OnClick eseménykezelő eljárást.

```
procedure Tform1.OnClick (Sender: TObject);
var
  BeírtAdat: String;
  OKLenyomva: Boolean;
begin
  BeírtAdat := '';
  OKLenyomva := InputQuery ('Adatbeviteli ablak',
  'Ide írjon valamit', BeírtAdat);
  if. OKLenyomva then
    ShowMessage ('A következő szöveget írta be:' +
    BeírtAdat);
end;
```

■ *Több formot tartalmazó alkalmazások*

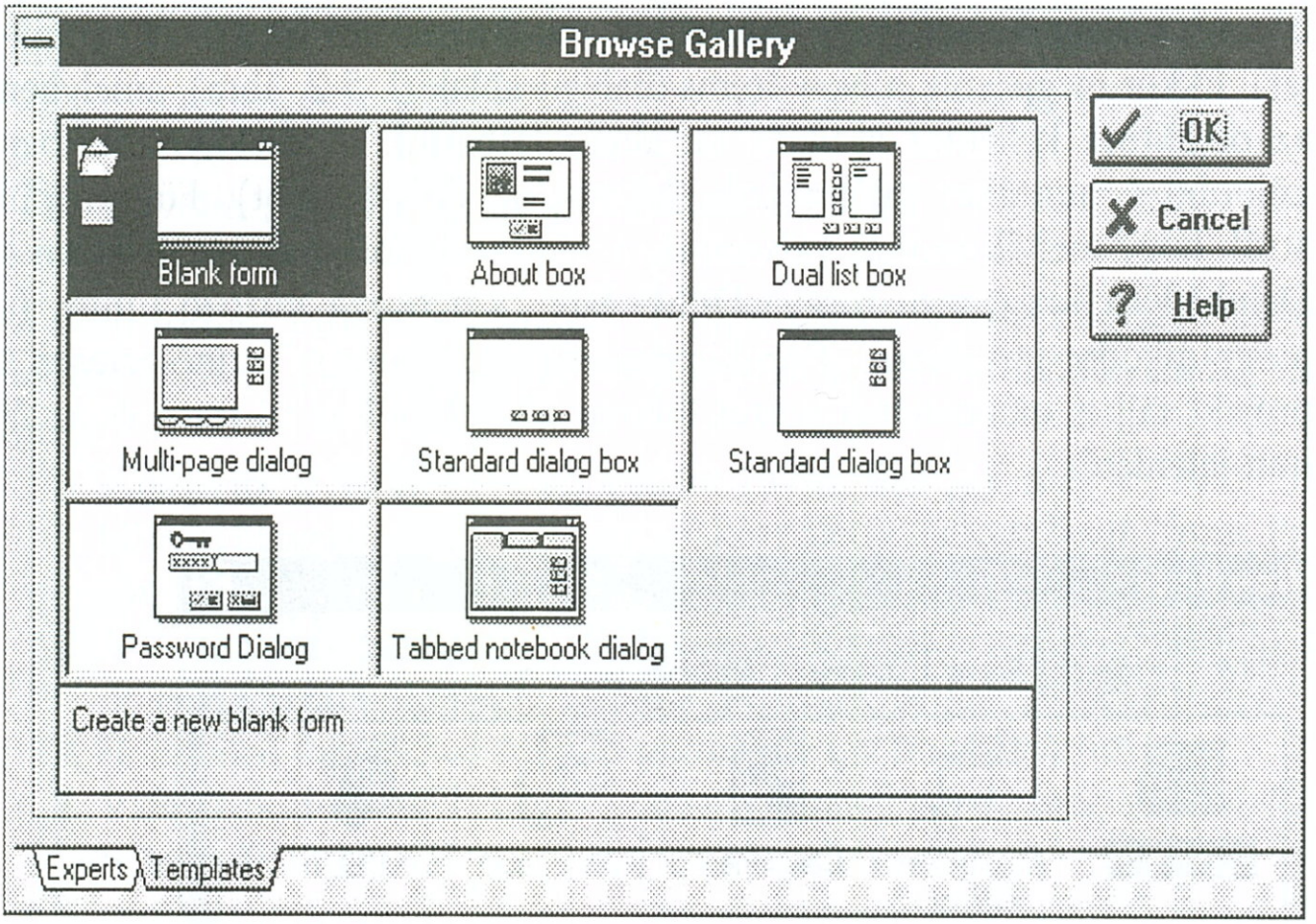
Idáig a projekteink csak egy formot használtak. Ebben. a fejezetrészben megismerkedhetünk a több formot használó (multiform) projektek készítéséhez szükséges technikákkal. Az alapelgondolás mindkét Delphi változatban ugyanaz, de a megvalósítás egy kicsit különböző.

Megjegyzés: *Az MDI (multiple document interface) modellt követő projektek létrehozásához a 13. fejezetet tanulmányozása szükséges.*

További formok illesztése Delphi 16 projektekhez

Induljunk ki abból az alaphelyzetből, hogy egy éppen elkezdett üres projekthez további formot kívánunk adni. Nyissuk meg a File menüt, és válasszuk a New Form pontot (vagy használjuk az eszközpaletta New Form gombját). Ekkor kinyílik az 5-1. ábrán látható Browse Gallery ablak. (Ha nem, akkor az Environment Options dialógusablak Gallery rész-

legében ellenőrizzük, hogy a Use on New Form opció ki van-e jelölve.) Az 5-1. ábrán látható, hogy a viszonylag gyakran használt form minták közül egyszerűen választhatunk (itt található az üres form is). Függetlenül attól, hogy melyik mintát választjuk, a választott formnak megfelelően a Delphi létrehoz egy új form (.DFM) fájlt és a hozzá tartozó unitot (a programkódot tartalmazó .PAS kiterjesztésű állományt). Természetesen a projekt fájl is módosításra szorul, hiszen jelezni kell az új form és az új unit jelenlétét.



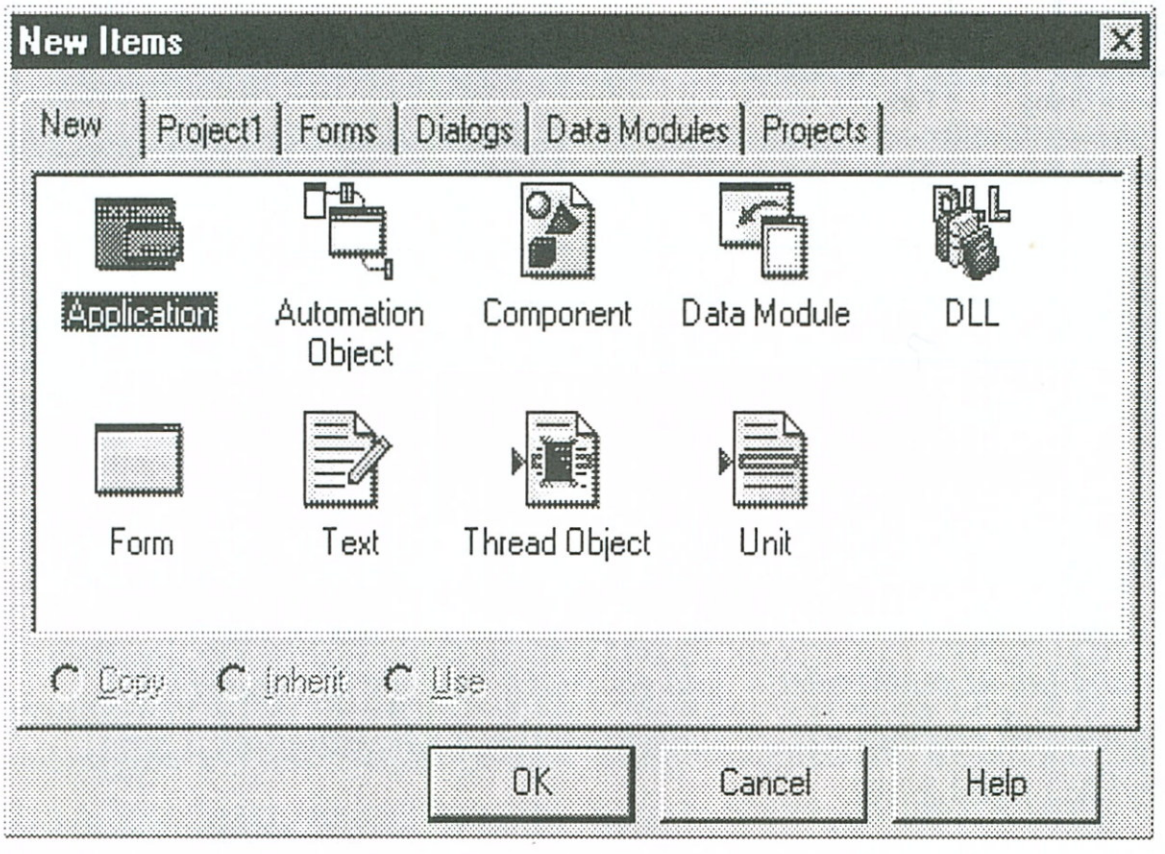
5-1. ábra. A Browse Gallery

Megjegyzés: Lehetőség van arra is, hogy egy már meglévő formot adjunk projektünkhöz. Ehhez válasszuk a File | Add File pontot (vagy használjuk az eszközpaletta Add gombját), ami egy dialógusablak megjelenését eredményezi. A dialógusablakban választhatunk a form (.DFM) fájlok közül.

Amikor egy meglévő formot adunk a projekthez, akkor a Delphi elvárja, hogy a formot vagy az ahhoz tartozó kódot használó összes unit **uses** sorában szerepeltessük a form nevét. A Delphi automatikusan kiegészíti a projekt (.DPR) fájl **uses** blokkját a szükséges hivatkozásokkal.

További formok illesztése Delphi 32 projektekhez

Induljunk ki abból az alaphelyzetből, hogy egy éppen elkezdett üres projekthez további formot kívánunk adni. Nyissuk meg a File menüt és válasszuk a New pontot (vagy használjuk a eszközpaletta New Form gombját). Ekkor megjelenik a New Items dialógusablak (l. 5-2. ábra), amelyben az objektum kelléktár aktuális tartalmát láthatjuk. A dialógusablakban különböző mintákat találhatunk (köztük az üres formot). Függetlenül attól, hogy melyik mintát választjuk, a választott formnak megfelelően a Delphi létrehoz egy új form (.DFM) fájlt és a hozzá tartozó unitot (a programkódot tartalmazó .PAS kiterjesztésű állományt). Természetesen a projekt fájl is módosításra szorul, hiszen jelezni kell az új form és az új unit jelenlétét.



5-2. ábra. A New Items a dialógusablak

Megjegyzés: Lehetőség van arra is, hogy egy már meglévő formot adjunk projektünkhöz. Ehhez válasszuk a File | Add File pontot (vagy használjuk az eszközpaletta Add gombját), ami egy dialógusablak megjelenését eredményezi. A dialógusablakban választhatunk a form (.DFM) fájlok közül.

Úgy, mint a Delphi 16-ban, amikor egy meglévő formot adunk a projekthez, a Delphi elvárja, hogy a formot vagy az ahhoz tartozó kódot használó összes unit **uses** sorában szerepeltessük a form nevét. A Delphi automatikusan kiegészíti a projekt (.DPR) fájl **uses** blokkját a szükséges hivatkozásokkal.

Több formot tartalmazó projektek kezelése

A több formot tartalmazó projekt formjai között, illetve a formok és az unitok között a Delphi mindkét változatában az itt leírt módon mozoghatunk:

1. Válasszuk a View | Forms pontot (használhatjuk még a SHIFT-F12 billentyűkombinációt vagy az eszközpaletta Select Form from list gombját).
2. A View Form dialógusablakban válasszuk ki azt a formot, amelyikkel dolgozni kívánunk, majd kattintsunk az OK gombra.

A különböző formokhoz tartozó unitokhoz így férhetünk hozzá:

1. Válasszuk a View | Units pontot (használhatjuk még a CTRL-F12 billentyűkombinációt vagy az eszközpaletta Select Unit from list gombját).
2. A View Unit dialógusablakban válasszuk ki azt a unitot, amellyel dolgozni kívánuk, majd kattintsunk az OK gombra.

Formok mentése

Formjainkról mindig célszerű másolatot készíteni. Ezt úgy tehetjük meg a legegyszerűbben, hogy a formot az eredetin kívül, egy másik név alatt is elmentjük a File | Save File As menüpontban. Az ilyen módon archivált példányokra soha ne gondoljunk úgy, mint kidobott háttértárkapacitásra.

A form elmentésekor csak a unit fájljának (.PAS) neve jelenik meg. Nem kell aggódnunk, a Delphi a form unit fájlja mellett magát a form fájlt is elmenti.

Form minták használata

Ha terveztünk egy komplikált formot, amelyet később is használni kívánunk, akkor valamilyen módon biztosítanunk kell, hogy az a Delphi számára elérhető maradjon. Ezt úgy tehetjük meg, hogy a formot felvesszük az ún. *form minták* (vagy *form sablonok*) közé. Ezután a formunk mindig elérhető lesz az új form beillesztésénél megjelenő Browse Gallery vagy Object Repository dialógusablakban attól függően, hogy a Delphi 16-ot vagy a Delphi 32-t használjuk.

A formot célszerű elmentenünk (a Delphi 32-ben kötelező), mielőtt felvennénk a form minták közé. Ezután:

1. Nyissuk meg a form gyorsmenüjét és
   - a Delphi 16-ban válasszuk a Save As Template pontot.
   - a Delphi 32-ben válasszuk az Add to Repository pontot. (Ez a Save Form Template dialógusablakot hozza elő.)
1. Megjelenik egy dialógusablak. A Title szerkesztési sorban adjuk meg mintánk nevét (érdemes ezen kívül egy rövid leírást is mellékelni a Description területen).

Egy bittérképet is mellékelhetünk sablonunkhoz, ha a Browse gombra történő kattintás után kiválasztjuk a kívánt (.BMP) fájlt.

Megjegyzés: *Lehetőség van a form fájl DLL-lé (dinamikusan szerkeszthető könyvtárrá) alakítására. L. a beépített help „Building a DLL". oldalát.*

Form (és a hozzá tartozó unit) törlése a projektből

A form (.DFM) fájl és a hozzá tartozó unit fájl, amely a kódot tartalmazza elválaszthatatlanok egymástól. Ennek köszönhetően az egyiket csak a másikkal együtt lehet eltávolítani. Amikor egy formot törlünk a projekt-

ből, a Delphi automatikusan törli a projekt (.DPR) fájl uses szakaszából a törlendő form unit fájljára utaló részt.

Kétféleképpen is eltávolíthatjuk a formot és a hozzá kapcsolódó unit fájlt a projektből:

- Jelöljük ki a törölni kívánt unitot vagy unitokat, és válasszuk a Remove file from project gombot az eszközpalettán.
- Jelöljük ki a törölni kívánt unitot vagy unitokat, és válasszuk a Remove File gombot az eszközpalettán.

(Egy további lehetőséget jelent a törlendő unit kijelölése után a File | Remove File pont választása a Delphi 16 File menüjében, illetve a File | Remove pont választása a Delphi 32 Project menüjében.)

A projekt főformjának kijelölése

Amikor egy több formot használó projekttel dolgozunk, módunkban áll kijelölni, hogy melyik legyen az a form, amelyet a Delphi az alkalmazás elindításakor automatikusan megjelenít. Ezt a formot *főformnak* nevezzük (main form). A többi form megjelenítése a program feladata (l. később).

Megjegyzés: *Amikor a felhasználó (például az ALT-F4 lenyomásával) vagy a program bezárja a főformot, az alkalmazás leáll. (A fejlesztői környezetből indított program leállításának is ez a legjobb módja.) Más szóval, a főform bezárása megegyezik az Application.Terminate hívás hatásával.*

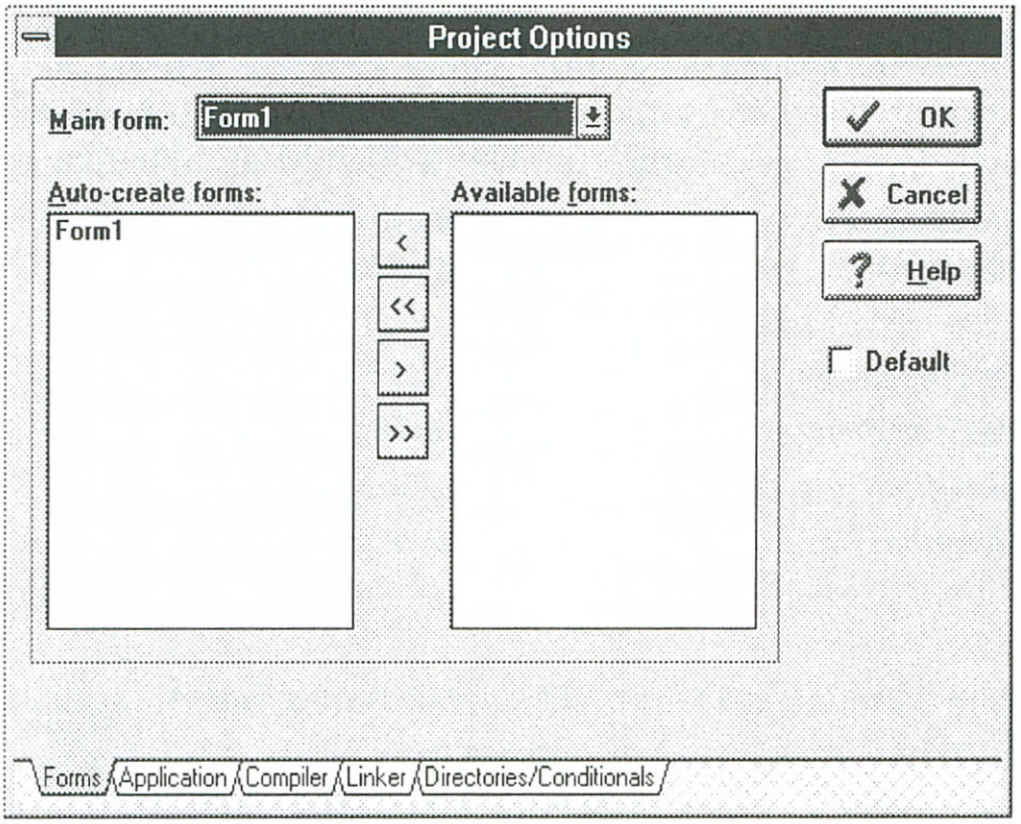
Az első létrehozott formnak a Delphi automatikusan a főform státust adja. Ezt fejlesztési időben így változtathatjuk meg:

1. Nyissuk meg a Project Options dialógusablakot (a Delphi 16-ban a az Options, a Delphi 32-ben a Project menüből érhető el). Az 5-3. ábrán a Delphi 16-beli dialógusablakot láthatjuk (a Delphi 32-ben található nagyon hasonló ehhez).
2. A Forms oldal Main Form legördülő listaablakában jelöljük ki azt a formot, amelyet főformnak kívánunk megtenni, majd kattintsunk az OK gombra.

Formok viselkedése futási idő alatt

Az 5-3. ábrán szembetűnik a két ablak: az „Auto-create forms" és az „Available forms" feliratú. Az Auto-create ablakban felsorolt formok automatikusan létrejönnek futási időben (automatikusan létrehozott formok). A 9. fejezetben foglalkozunk azon formok kezelésével, amelyeket nem szeretnénk ha a Delphi automatikusan létrehozna futási időben. Alapértelmezésben minden form automatikusan létrehozott.

Megjegyzés: *Még arra is lehetőség van, hogy egy olyan formot hozzunk létre futási időben, amelyet fejlesztési időben nem is adtunk a projekthez. A 9. fejezetben ezt is megtanulhatjuk.*



5-3. ábra. A project Options dialógusablak

Ha el szeretnénk érni, hogy az egyik form használhasson egy másik formot és annak komponenseit, akkor az előbbi **uses** szakaszában fel kell tüntetni az utóbbi unit fájljának nevét, mert ezt a Delphi nem végzi el helyettünk. (A Delphi 32-ben azonban ezt a műveletet a File | Use Unit választással egyszerűbbé tehetjük.) Ezzel lehetővé válik az, hogy az egyik formhoz tartozó kódban a másik form objektumait és tulajdonságait hasz-

náljuk. Tekintsünk példaként két formot. Legyen ezek neve Form1 és Form2 és a hozzájuk tartozó unit (.PAS) fájlokat nevezzük rendre Unit1-nek és Unit2-nek. Tegyük fel továbbá, hogy mindkét form rendelkezik egy-egy szerkesztési ablakkal (Form1EditBox és Form2EditBox).

Ha a Form2 unitjába az alábbi sort írjuk, akkor a Form1EditBox tartalma megjelenik a Form2EditBox szerkesztési ablakban.

```
Form2EditBox.Text := Form1.Form1EditBox.Text;
```

(Figyelem! Ha a Unit2 **uses** szakaszában nem tüntetjük fel a Unit1-et, akkor a program nem működik.)

Ha egyszer elértük a **uses** szakasz megfelelő unit névvel történő kiegészítésével, hogy az egyik form felismerje a másikat, akkor az utóbbi tulajdonságira és objektumaira való hivatkozáskor fel kell tüntetni a form nevét, egy pontot, és csak azután következik a tulajdonság vagy az objektum neve. (Az előző példában nem kellett volna használnunk a másik formnevét, ha a saját unitjában írjuk a programot.)

Megjegyzés: Ha el szeretnénk érni, hogy két form egymás objektumaival dolgozhasson, és ezért mindkét form ***uses*** *szakaszába beiktatjuk a másik unit fájljának nevét, akkor előfordulhat, hogy a fordító a kölcsönös hivatkozás miatt hibaüzenetet ad. Ezt úgy kerülhetjük el, ha mindkét form unit fájljának implementációs blokkjában, a* ***uses*** *szakaszban feltüntetjük a másik form nevét.*

(Ha egy unitnak nincs **uses** szakasza az implementációs blokkban, akkor azt nekünk kell pótolnunk: egyszerűen írjuk a **uses** kucsszót az **implementation** kulcsszó alá.)

A másodlagos (nem fő-) formok egészen addig láthatatlanok a felhasználó számára – még akkor is, ha azok automatikusan jöttek létre –, amíg azt a programban meg nem változtatjuk. Itt közöljük azoknak a metódusoknak a rövid leírását, amelyek a formok futási időben történő kezeléséhez szükségesek.

Show. A Delphi által már létrehozott formot a Show metódussal tehetjük láthatóvá és aktívvá is egyszerre.

Például ha a Form2-t elérhetővé kívánjuk tenni a Form1-ből:

1. Adjuk a Form1 **uses** szakaszához a Unit2-t.
2. Futassuk le az alábbi kódrészletet: Form2.Show.

Tipp: *Ne felejtsük el a FormStyle tulajdonságot fsStayOnTop-ra állítani, ha szeretnénk, hogy egy adott form mindig a többi ablak fölött legyen.*

Hide, Close. A Hide metódus egyszerűen elrejti a formot, nem törli azonban a memóriából. Ha a memóriából is törölni kívánjuk, akkor hívjuk meg a Close metódust (ami megegyezik a rendszermenü Close pontjának vagy az ALT-F4 hatásával).

Modális formok

Néha szükségünk lehet egyes formok modálissá tételére. A modális formokat az jellemzi, hogy azokat mindenképpen be kell zárni, mielőtt lehetőségünk lenne más formmal dolgozni (ilyenné tehetünk például egy saját tervezésű dialógusablakot).

Megjegyzés: *A Windows-konvenció szerint a dialógusablakok nem tartalmaznak maximalizáló és minimalizáló ikonokat, és nem méretezhetőek át.*

(Emlékezzünk vissza: ha egy formot ilyenné akarunk tenni, akkor a különböző BorderIcons tulajdonságokat False-ra, a BorderStyle tulajdonságot pedig bsDialog-ra kell állítani.)

Ha egy formot modálissá kívánunk tenni, akkor a Show metódus helyett használjuk a ShowModal metódust. Szintaxisa így néz ki:

FormName.ShowModal;

vagy

if *FormName*.ShowModal = mrOK then....

(Megjegyzés: a ShowModal metódus megállítja a program futását, míg a Show nem.)

A metódus hívásával láthatóvá tett formmal a felhasználónak kapcsolatba kell lépnie. Ha a felhasználó nem zárja be a formot, akkor az a képernyőn marad, és a program megreked.

Ahhoz, hogy a Delphi bezárjon egy modális formot az kell, hogy annak ModalResult tulajdonságát előzetesen nullától különböző értékre állítsuk. Az ezt megváltoztató kódot a form vagy egy komponens bármely eseménykezelő eljárásában elhelyezhetjük. A szabályos módszer azonban az, hogy a modális formra nyomógombokat helyezünk vagy esetleg egy OnTimer eseménykezelőt használunk. Az előbbi esetben a gombok ModalResult tulajdonságát a gomboknak megfelelően állítjuk be. Például OK gomb használatakor állítsuk a gomb ModalResult értékét mrOK-ra. Amikor a felhasználó a modális form egy gombjára kattint, amelynek a ModalResult tulajdonsága nullától különböző (mrOK, mrCancel, mrAbort stb.) értékre van állítva, akkor a Delphi ezt az értéket bemásolja a form ModalResult tulajdonságába, azaz a form bezáródik.

Tipp: *A TBitBtn komponens rendelkezik egy Kind nevű tulajdonsággal, amelynek használata könnyebbé teszi ezt az eljárást.*

Programozás formokkal

Különösen amikor több formot használó alkalmazásokkal dolgozunk, lényeges, hogy számoljunk azzal az eshetőséggel is, hogy minden form el van rejtve – de az alkalmazás még fut! (Természetesen módunkban áll a formok elrejtését megakadályozni az OnHide esemény segítségével, itt azonban én főleg az olyan szituációkra gondolok, amikor a főform rejtetté válik rögtön az elején, és a többi formot pedig a felhasználó bezárja a Windows-on keresztül (pl. az ALT-F4 billentyűkombinációval).

Ilyen esetben használható jól az OnCloseQuery és az OnClose esemény. Segítségükkel megvizsgálhatjuk a formok bezárásának okát. Ebben a fejezetrészben ezt a két eseményt mutatjuk be.

Az OnCloseQuery esemény

Az OnCloseQuery esemény akkor következik be, amikor a felhasználó megkísérli bezárni a formot. Ezért az OnCloseQuery eseménykezelője az a hely, ahol a form bezárását megakadályozhatjuk. Ezt az eseményt használjuk például akkor, amikor a form bezárása előtt meg akarjuk kérdezni

a felhasználót, hogy valóban ki akar-e lépni a formból. Az OnCloseQuery első sora így néz ki:

```
procedure TForm1.FormCloseQuery (Sender: TObject; var
CanClose: Boolean);
```

Ha a CanClose paraméter értékét False-ra változtatjuk az eseménykezelőn belül, akkor a form nem lesz bezárható! Például:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
var CanClose: Boolean);
  begin
    if MessageDlg('Valóban be akarja zárni a
    formot?', mtConfirmation, [mbOK, mbCancel], 0) =
    mbCancel then
    CanClose := False;
  end;
```

Az OnClose esemény

Az OnClose esemény az utolsó lehetőség arra, hogy egy form bezárását megakadályozzuk (nem úgy mint a Visual Basicben). A legtöbbször azonban a Delphi programozók ezt az eseményt a memória felszabadítására használják. Az OnClose esemény Action paramétere négy különböző értéket vehet fel. Ezeket az alábbi táblázat tartalmazza:

Action. paraméter értéke	*Jelentése*
caNone	A formot nem engedi bezárni, hatása megegyezik az OnCloseQuery esemény CanClose paraméterének False-ra állításával.
caHide	A formot elrejti (ahelyett, hogy bezárná).
caFree	A form bezáródik, a számára lefoglalt memória teljes egészében felszabadul.
caMinimize	A form nem záródik be, hanem minimalizálódik. (Az MDI gyermekformjainak alapbeállításában is ez szerepel – l. 13. fejezet.)

6. FEJEZET

Magasabb szintű programozói technikák

Az előző fejezet a Delphi Object Pascal alapjait mutatta be. Ez a fejezet egy következő lépcsőfokot jelent a Delphi tanulásában. A fejezet elején megtanulhatjuk, hogyan lehet kiegészíteni a Delphi beépített típusait saját típusok létrehozásával. Majd áttérünk olyan típusok tárgyalására, amelyek segítségével egyszerre több adattal is dolgozhatunk. Ilyenek a listák, a tömbök és a rekordok. Ezek után ízelítőt kaphatunk a mutatók használatáról és a memóriakezelésről. Ezt követi a Delphi beépített függvényeinek és eljárásainak ismertetése, majd megtanulhatjuk, hogyan írhatunk magunk ilyeneket. A fejezet végén a projektek és unitok felépítésével, a goto utasítással és azzal foglalkozunk, hogy miként használhatjuk minél hatékonyabban a Delphi fordítóját, amikor különálló programokat kívánunk létrehozni.

■ *Újra a típusokról*

A Delphi programozási nyelve a szoros típusmegkötésű nyelvek közé tartozik. Ez azt jelenti, hogy minden változónak kell hogy legyen típusa. A Delphi 16-ban a típushatárok meglehetősen szigorúak; a Delphi 32 új Variant típusának használatával ezek átjárhatóvá válnak. Mindazonáltal marad néhány korlát: függetlenül attól, hogy a Delphi melyik verzióját használjuk, ha egy változót például Double típusúnak deklarálunk, akkor azt nem adhatjuk értékül egy egész típusú változónak. Másfelől azonban egy sztringváltozónak adhatunk karaktert értékül (l. előző fejezet), vagy egy lebegőpontos változóba bemásolhatjuk egy egész típusú változó értékét. Lehetőség van két valós típusú változó vagy két egész típusú változó közti adatmozgatásra is, ha az értékek kompatíbilisek.

Továbbá (néhányan hozzátennék, hogy „sajnos"), ha a Variant típussal dolgozunk, módunkban áll egy sztringet tartalmazó Variant változóhoz hozzáadnunk egy számot tartalmazó Variant változót anélkül, hogy hibaüzenetet kapnánk.

Megjegyzés: A beépített súgó „Assignment Compatibility" és „Type Compatibility" címszavai alatt megtalálhatjuk a pontos értékadásfajtákat, amelyek a Variant típussal megvalósíthatók.

Tapasztalatból tudom, hogy a hibakeresés egyszerűbb (és a futás is gyorsabb) azon programok esetén, amelyekben a Variant típust csak végső eszközként használjuk, amikor annak speciális tulajdonságaira van szükség (általában OLE alkalmazásokban – l. 13. fejezet). Vannak azonban programozók, akik a Delphi típus megkötéseit kényszerzubbonynak tartják, és kimondottan örülnek a Variant típus megjelenésének a Delphi 32-ben. A döntést az olvasóra bízom.

Sorszámozott típusok

A Delphi beépített típusainak – mint pl. sztring és egész – kombinálásával és módosításával új típusokat hozhatunk létre. A lehetőségek szinte korlátlanok. Az új típusok előállításának módszerei közül először azokkal ismerkedünk meg, amelyekkel *sorszámozott típusokat* hozhatunk létre. Annak ellenére, hogy a könyvben most bukkan fel először ez a kifejezés, az ebbe a kategóriába tartozó leggyakoribb típusokkal már találkoztunk. Ilyen pl. minden egész típus és a Char is. A sorszámozott típusokat az alábbi jellemzők definiálják:

- Véges sok elemből állnak.
- Valamilyen módon rendezettnek kell lenniük.

A kettő együtt azt jelenti, hogy minden elemnek, kivéve a maximálisat (ha van), van pontosan egy rákövetkezője. A valós típusok nem sorszámozottak, hiszen ott nincs értelme a rákövetkezésnek azaz a „Mi az 1 után következő valós szám?"-hoz hasonló kérdések értelmetlenek. Mindkét Delphi verzióban az alapvető sorszámozott típusok a következők: a Boolean, a Char és minden egész típus.

A sorszámozott típus műveletei

A Delphi mindkét verziójában ugyanazok a sorszámozott típusokat kezelő beépített függvények találhatók. A 6-1. táblázat ezeket foglalja össze.

6-1. táblázat. Sorszámozott típusokra alkalmazható függvények

Függvény	*Hatása*
Pred(X)	Az X-et megelőző elemet választja ki.
Succ(X)	Az X-et követő elemet választja ki.
Dec(X; n)	Megadja az n hellyel az X előtt álló elemet.
Inc(X; n)	Megadja az n hellyel az X után álló elemet.
Ord(X)	Megadja az elem indexét.

Megjegyzés: *Ha a műveletek kilépnének az adott típusból (pl. ha a Succ-ot az utolsó elemre alkalmazzuk), és az értéktartomány ellenőrzés be van kapcsolva, akkor a Delphi futási idejű kivételt (exception) vált ki. (A kivételkezelést ld. a 8. fejezetben.)*

Felsorolt típusok

Új sorszámozott típusok előállításának a legegyszerűbb módja az, hogy valamilyen sorrendben elemeket sorolunk fel. Minden elemet egy azonosítóval adunk meg. (Ez azt jelenti, hogy meg kell felelniük az azonosítókra vonatkozó szabályoknak.) Ezzel a technikával ún. *felsorolt típusokhoz* juthatunk. Íme néhány példa a felsorolt típusok létrehozására:

```
type
  Cég = (Compaq, Dell, Gateway, Packard, Micron,
  Zeos, Más);
  MunkaNapok = (hétfő, kedd, szerda, csütörtök,
  péntek);
```

Ha egyszer létrehoztuk a kívánt típust, akkor lehetőség van az ilyen típusú változók deklarálására a megszokott szintaxis használatával. Egy a fent definiált Cég típust használó deklaráció a:

```
var
  SzámítógépMárka: Cég;
```

A felsorolt adattípus elsődleges feladata jelezni a számítógép számára, hogy milyen adatokkal kell majd műveletet végeznie. A felsorolt adattípusra jellemző, hogy

- nem tartalmaz 256 elemnél többet,
- minden egyes elemnek külön azonosítója van.

A felsorolással egyben megadunk egy rendezést is. A felsorolás sorrendjének megfelelően az elemek indexeket kapnak. Ezeken az indexeken keresztül kezeli a Delphi a típus elemeit. Az indexezés mindig 0-tól kezdődik (és nem 1-től!). Például az előző Cég típusban a Compaq indexe 0 lesz, a Más a 6-ot kapja. A 6-1 táblázat sorszámozott típusra vonatkozó függvényeivel a felsorolt típus elemei között is mozoghatunk.

Intervallumtípusok

Már az intervallumtípusra is láttunk példát a case utasítás tanulmányozásánál (l. 5. fejezet). Az 5. fejezetbeli példaprogram egy részlete így nézett ki:

```
case PontÁtlag of
  90..99: {utasítás};
```

A 90 . . 99 egy intervallumot ír le, amely 90-nél nem kisebb és 99-nél nem nagyobb egészeket tartalmaz. Intervallumokat általánosan úgy definiálhatunk, hogy az intervallum határait megadó sorszámozott típus (lehet előzőleg definiált felsorolt típus is) két eleme közé két pontot teszünk. Például:

```
type
  VezérlőKarakterek = #0 . . #31; {a # jel után a
  karakter ASCII kódja áll}
  Nagybetűk = #65 . . #90;
```

Halmazok

A halmazok hasonlóan működnek, mint a felsorolt típusok, de azok nem sorszámozottak, mivel itt az elemek rendezettsége nem lényeges. Halmaz típust akkor használunk, ha arra vagyunk kíváncsiak, hogy milyen elemek tartoznak egy bizonyos csoporthoz, de az elemek sorrendje nem érdekel minket. A felsorolt típusokhoz hasonlóan a halmazok is legfeljebb 256 elemből állhatnak, *de minden elem ugyanahhoz a sorszámozott típushoz kell hogy tartozzon, továbbá ha az elemek numerikusak, akkor értékük 0 és 255 közé kell hogy essen.* Nem készíthetünk olyan halmazt, amelyben egyaránt vannak karakterek és egészek, valamint olyat sem, amelynek eleme pl. a 20 000. Egy halmazdeklaráció így néz ki:

```
type
  Karakterek = set of Char;
  Prímek = set of Byte;
```

Ha a halmazt elemeivel kívánjuk felsorolni, használjunk szögletes zárójelet, és az elemeket vesszővel válasszuk el egymástól.

```
var
  KisPrímek: Prímek;
  AngolKisMagánh, AngolNagyMagánh, AngolMagánh:
  Karakterek;
begin
  KisPrímek := [2, 3, 5, 7, 11, 13, 17];
  AngolKisMagánh := ['a', 'e', 'i', 'o', 'u'];
```

A halmazokat kezelő műveleteket a 6-2. táblázat foglalja össze. A halmazműveletek működését az alábbiak illusztrálják:

```
AngolMagánh := ['a', 'A', 'e', 'E', 'i', 'I', 'o',
'O', 'u', 'U'];
```

továbbá

```
AngolKisMagánh := ['a', 'e', 'i', 'o', 'u'];
AngolNagyMagánh := ['A','E','I','O','U'];
```

6-2. táblázat. A halmaz típus műveletei

Művelet	*Hatása*
= (egyenlőség)	Megvizsgálja, hogy a két halmaz egyenlő-e. Egy logikai (True/False) eredményt ad vissza.
<> (nem egyenlőség)	Megvizsgálja, hogy a két halmaz különböző-e. Egy logikai (True/False) eredményt ad vissza.
<= (tartalmazás)	Ellenőrzi, hogy az egyik halmaz összes eleme megtalálható-e a másikban (az egyik részhalmaza-e a másiknak). Szintén logikai (True/False) eredményt ad vissza.
>= (fordított tartalmazás)	Ellenőrzi, hogy a másik halmaz összes eleme megtalálható-e az egyikben.
+ (unióképzés)	Egy olyan halmazt ad eredményül, amely mindkét halmaz összes elemét tartalmazza.
* (metszetképzés)	Egy olyan halmazt ad eredményül, amelyet a mindkét halmazban egyszerre megtalálható elemek alkotnak.
- (különbségképzés)	Az egyik halmaz elemeiből kiveszi a másik halmaz elemeit.
in (elemvizsgálat)	Ellenőrzi, hogy a megadott változó értéke eleme-e a halmaznak. Egy logikai (True/False) eredményt ad vissza.

ekkor a halmazokra igaz, hogy

AngolMagánh = AngolKisMagánh + AngolNagyMagánh
AngolKisMagánh = AngolMagánh - AngolNagyMagánh

Halmazok használatával kikerülhetjük a nehézkes case utasításokat, így olvashatóbbá és gyakran rövidebbé is tehetjük programjainkat. Például az

```
if BekértKarakter in AngolMagánh then ...
```

sor tisztább és rövidebb is, mint az ugyanezt megvalósító **case** utasítás.

■ Tömbök

Tömböket (array) használunk, amikor összefüggő értékek egy olyan csoportjával kívánunk dolgozni, amely számára szükséges előre helyet biztosítani a memóriában. A tömbök segítségével oldható meg a nagy sztringek, listák valamint táblázatok tárolása is. A tömbök az első példái annak, amit az Object Pascalban strukturált típusnak nevezünk. Egy strukturált típust az jellemez, hogy nemcsak a típussal, mint egésszel végezhetünk műveleteket, hanem lehetőség van arra is, hogy az egyes elemekkel külön-külön dolgozzunk. Ilyennel a könyvben még nem foglalkoztunk. Egy strukturált típus a Delphi 16-ban maximum 65 520 byte-ot foglalhat el, míg a Delphi 32-ben akár 2 gigabyte-ot (!) is.

Egydimenziós tömbök

Az egydimenziós tömb (vagy lista) egyszerűen értékek egy speciális listájának tekinthető, ahol az elemeket az alábbi két dolog jellemzi:

- a tömb neve
- az elem pozíciója (sorszáma) a listán

Tegyük fel, hogy egy olyan programot szeretnénk írni, amelyben minden nap elején a felhasználó megadhatja a napi teendőit. Ezeket az információkat érdemes általában listán tárolni. Nevezzük a listát pl. Teendőknek. Ekkor az első elemre Teendők[1]-gyel, a másodikra Teendők[2]-vel hivatkozhatunk és így tovább.

A zárójelek közti számot indexnek hívjuk. Ne tévesszen meg senkit, hogy matematikában az indexeket a betű jobb sarkában elhelyezett kis számmal jelezzük: M_5, a Delphiben ugyanez M[5]-ként írandó. Az Object Pascalban az M5 egy egyszerű változónév, de az M[5] az M nevű tömb egyik (ötödik) elemét jelöli.

Visual Basic Tipp: *Fontos, hogy az indexeket a Delphiben mindig szögletes (és ne kerek!) zárójelek közé tegyük.*

A tömböket használat előtt deklarálni kell. Az itt közölt példában két tömböt (az egyik sztringeket tartalmaz, a másik egészeket) deklarálunk:

```
var
  USAÁllamnevek: array[1..50] of String;
  EgészTömb: array [-100..100] of Integer;
```

Ez azonban nem a legszerencsésebb módja a tömbök deklarálásának. Nagyobb programokban célszerű minden tömbtípust előbb külön deklarálni ahelyett, hogy a típusdeklarációkat implicit módon tegyük meg a változó deklarációk részeként. Ez teszi lehetővé azt, hogy ezeket a tömböket paraméterként adhassuk át alprogramok hívásakor (l. később). Például:

```
type
  VálasztásiEredmények = array[1..51] of Longint;
var
  Eredmények96: VálasztásiEredmények;
```

Egy egydimenziós tömb deklarálásának általános szintaxisa így néz ki:

TömbNév: array[*Indextípus*] of *Alaptípus*;

ahol az Indextípus bármilyen sorszámozott (vagy intervallum) adattípus lehet, kivéve a Delphi 16 Longint típusát. (A Delphi 32-ben használhatjuk a Longint-et is.)

Amikor egy fix tömbbel kívánunk dolgozni, azt érdemes egy típusos konstans formájában deklarálni. A típusos konstans lehet ilyen bonyolult típusú is, hasonlóan más statikus változókhoz. A deklaráció a **const** szakaszban történik. Például:

```
Const
  PontHatárok: array[1..4] of Integer = (90, 80, 70, 60);
```

(Vegyük észre, hogy a kezdeti értékek megadásához kerek zárójelet használtunk és a tömböt a **const** részben hoztuk létre.)

A Delphi 16 sztringjei igazából karaktertömbök. Egy adott sztring tömbjének kezdőindexe 1, záróindexe pedig a karaktersorozat hosszával egyenlő. (A nulladik pozícióban a Delphi a karakterekek számát tárolja.) Például a

```
var
  S: String[80] ;
```

részletben deklarált S változót a Delphi valójában az alábbi karaktertömb formájában tárolja:

```
S: array [0..80] of Char;
```

A Delphi 16-ban (amennyiben a Delphi 16-tal kompatíbilis sztring típust használunk, akkor a Delphi 32-ben is) a sztringeket leíró tömbök 0. pozíciójában a karakterlánc hossza tárolódik. A tulajdonképpeni sztring első karaktere az S[1]-ben található, a második az S[2]-ben és így tovább. Mivel a Delphi nem tesz különbséget a sztring és az annak megfelelő tömb között, ezért a sztring egyes karaktereire ugyanolyan módon hivatkozunk, mint egy tömb egyik elemére. Például az alábbi programsor az S sztring 37. karakterét változtatja felkiáltójellé.

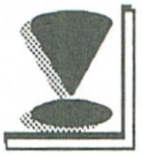
```
S[37] := '!';
```

A Delphi 16 sztringhossz-korlátozását részben megkerülhetjük, ha egy 255 eleműnél nagyobb karaktertömböt definiálunk. A későbbi „Nullavégű sztringek és a PChar" alfejezetben látni fogjuk, hogy akkor járunk a legjobban, ha olyan karaktertömböt használunk, amelynek indexezése nullától kezdődik (a nulla indexű elemnek itt is adminisztratív szerep jut). Tegyük fel, hogy egy 20 000 karakterből álló sztringre van szükségünk. A Delphi 16-ban ekkor a következőket használhatnánk:

```
var
  Szöveg: array[0..20000] of Char;
```

Ezzel a memóriában 20 001 karakter (közülük egy adminisztratív célokat szolgál) tárolására alkalmas helyet foglalunk. (Az ilyen tömbtípus

használatának az a hátránya, hogy a lokális heap-ből memóriát használunk el.)



Figyelem: Nagy sztringek használatánál (amelyek 255 karakternél hosszabbak) a Delphi 16-ban előtérbe kerül a memóriakezelés fontossága. Lásd a „Mutatók" szakaszt ebben a fejezetben.

A Delphi 32 a nagy sztringek által elfoglalt memóriát automatikusan felszabadítja, így ha ezzel a verzióval dolgozunk, nem kell aggódnunk a memóriakezelés miatt.

Sztringlisták

A Delphi mindkét változata nagyon sok komponenssel kapcsolatos adatot – többek között a listaablakok és a csatolt listaablakok tartalmát, az elérhető fontokat, vagy a memóablakok sorait – úgynevezett *sztringlista* formájában tárol. A sztringlisták lényegében sztringeket tartalmazó tömbök, amelyek egyszerű kezeléséhez a Delphi biztosít eszközöket.

Tipp: Azt, hogy melyik tulajdonság tárolódik sztringlistaként, egyszerűen megállapíthatjuk az objektumfelügyelőben. Ha egy tulajdonság neve mellett az érték oszlopban egy TStrings kezdetű értéket találunk, akkor az egy sztringlistában tárolt tulajdonság. (A TStrings a sztringlista objektum neve. A TStrings objektummal még foglalkozunk a 7. fejezetben is. Láthatjuk majd például, hogyan készíthetünk saját sztringlistákat futási időben.)

Egy sztringlistát egy másikba közönséges értékadással másolhatunk. Ha például egy memóablak sorait meg szeretnénk jeleníteni egy csatolt listaablakban, akkor az alább látható sorral elérhetjük, hogy a ComboBox komponens Items tulajdonsága arra a memóriarészre mutasson, ahol a listaablak szövege található!

```
ComboBox1.Items := Memo1.Lines;
```

Ha egy valódi másolatot szeretnénk:

```
ComboBox1.Items.Assign (Memo1.Lines);
```

(Mind a csatolt listaablak Items tulajdonsága, mind a memó komponens Lines tulajdonsága TStrings típusú.)

Tipp: A Delphiben a sztringlisták lehetőséget nyújtanak bizonyos objektumokból álló listák kezelésére is. Gyakori alkalmazása ennek, amikor a sztringlista minden eleméhez egy-egy képet kapcsolunk. A részleteket az olvasó a beépített súgó „String List Topic" pontja alatti „Adding Objects To A String List" és „Adding Graphical Objects To A String List" részekben találhatja.

Az alábbiakban a sztringlisták leggyakrabban használt tulajdonságait és metódusait ismertetjük.

Strings. Minden sztringlista rendelkezik Strings tulajdonsággal, amelyre legjobb ha úgy tekintünk, mint a tárolt sztringekből álló tömbre. A lista a 0 indexről indul. Például a memóablakok az egyes szövegsorokat a Lines nevű sztringlistában tárolják, a szöveg egy adott sorát a Strings tulajdonság megfelelő indexű elemével érhetjük el. A

```
Memo1.Lines.Strings[1]
```

a memókomponens második sorát jelenti. (A Memo1.Lines.Strings[0] tartalmazza az elsőt.)

Megjegyzés: Mivel a Strings a sztringlisták alapértelmezés szerinti tulajdonsága, ezért nevét nem szükséges feltüntetnünk. A fordító megérti az olyan sort is, mint Memo1.Lines[1] (a második sor a memókomponensben).

Count. A Count tulajdonság a listán szereplő sorok számát tartalmazza. Ez azt jelenti, hogy a lista utolsó sztringjének indexe Count-1. Például:

```
Memo1.Lines.Strings[Count-1] := 'Helló'
{értékadás az ablak utolsó sorának};
```

vagy

```
FontokSzáma := Screen.Fonts.Count;
{Megadja az elérhető képernyőfontok számát};
```

Mivel a sztringlisták nulla kezdőindexűek, elemein könnyen végigsöpörhetünk a következő sor használatával:

For *Index* :=0 to *StringList.*Count-1 do

IndexOf. Ez a függvény egy megadott sztring helyét keresi meg. Ha nem találja, akkor visszatérési értéke –1 lesz. Például:

```
If Memo1.Lines.IndexOf ('Ott vagyok') = -1 then
  ShowMessage ('A Sztringet nem találtam meg')
else
  ShowMessage ('Megtaláltam!');
```

Add. Az Add metódus a megadott sztringet a lista végéhez fűzi, ha a Sorted tulajdonság értéke False. Ha a Sorted tulajdonság True-ra van állítva, akkor a sztringet úgy illeszti a listára, hogy az abc-sorrend megmaradjon. A listaablak Items tulajdonsága egy olyan sztringlista, amely a listaelemeket tárolja. Ha a listához az amerikai elnökök nevét szeretnénk hozzávenni, akkor az ezt megvalósító kód részlete így nézne ki:

```
Elnök.ListBox.Items.Add ( 'George Washington' );
Elnök.ListBox.Items.Add ( 'John Adams' );
```

Az Add-ot függvényként is használhatjuk, ebben az esetben megkapjuk a beillesztendő sztring indexét a listán. Például:

```
UtolsóAdatIndex: = ListBox1.Items.Add ('Ezt írtam be
utoljára');
```

Insert. Az Insert metódus segítségével, ha a Sorted tulajdonság értéke False, akkor egy megadott helyre illeszthetjük a kívánt sort. Például, ha kifelejtjük a kilencedik elnököt (gyakori hiba, hiszen csak egy hónapig birtokolta a posztot), akkor az alábbi kódsort használnánk. (Ne felejtsük, a sztringlisták nulla kezdőindexűek, így a 9. elem a 8. index alatt található.)

```
Elnök.ListBox.Insert (8, 'William Henry Harrison');
```

Az első paraméter az index, ahová a sort be kívánjuk szúrni, a második pedig maga a beillesztendő sztring.

Move. A Move metódussal a sztringlista elemeit rendezhetjük át. Például a listaablak 10. sorának tartalmát így helyezhetjük az első sorba:

```
Elnök.Items.Move (9, 0) ;
```

Első paraméterként adjuk meg a helyet, ahol az áthelyezendő sztring áll, másodikként pedig a beillesztés helyét.

Clear, Delete. A Clear metódus a lista összes sztringjét törli. A Delete metódus a megadott indexű elemet törli a listáról. Például a

```
Elnök.ListBox.Items.Delete (0);
```

sor George Washington nevét törli.

Többdimenziós tömbök

Létrehozhatunk egynél több dimenziós tömböket is; ezeket hívjuk többdimenziós tömböknek. Ha az egydimenziós tömböket egy adatlistának képzeltük el, amelynek kezeléséhez egy index kellett, a kétdimenziós tömböt képzeljük egy adattáblázatnak, amelynek elemeire két index-szel hivatkozhatunk. Tegyük fel például, hogy egy szorzótáblát szeretnénk elhelyezni a memóriába, táblázat formájában. Ekkor egy ehhez hasonló programot írnánk:

```
var
  I, J: Integer;
  SzorzóTábla: array[1..12, 1..12] of Integer;
begin
  for I := 1 to 12 do
    for J := 1 to 12 do
      SzorzóTábla [I, J] := I*J;
end;
```

Egy többdmenziós tömb elemszámát úgy kapjuk, hogy összeszorozzuk az indextartományok (intervallumok) elemszámát. A fenti változó deklaráció 144 (12*12) elemnek foglal helyet. A tömb egy 12 sorból és 12 oszlopból álló táblázatnak (vagy mátrixnak) tekinthető.

Amikor kétdimenziós tömböt használunk egy mátrix ábrázolására, a matematikai konvenció szerint az első intervallum a sorok, míg a második az oszlopok számára utal. A fenti programrészletre alkalmazva ezt a konvenciót, azt mondhatjuk, hogy az az egyes sorokat tölti ki oszlopról oszlopra haladva. (Vegyük észre, hogy ez a konvenció az ellenkezője annak, amit a Delphi a táblázatok Cell tulajdonságában használ.)

Rekordok

Tegyük fel, hogy van 3 egydimenziós tömbünk, amelyekben egy cég 100 emberének adatait tároljuk. Az első tömbben a nevek vannak felsorolva, a második a fizetéseket tartalmazza, a harmadikban pedig a társadalombiztosítási azonosítójukat tároljuk. Ha külön-külön próbáljuk kezelni ezeket az összetartozó adatokat, rosszul járunk, párhuzamos mozgatásuk a memória két része között, vagy a memória és a háttértár között sok munkát jelent.

Ezt a pluszmunkát egy új struktúra az ún. rekord használatával kerülhetjük ki. A rekord nem más, mint egy olyan típus, amellyel a Delphi különböző típusú (beleértve az általunk létrehozott típusokat is) elemeit foghatjuk össze.

Az itt közölt programrészlet az egyik képzeletbeli alkalmazottunk rekordtípusát definiálja:

```
type
  TFontosInfo = record
    Név: String;
    Fizetés: Longint;
    TársAz: String [11];
end;
```

A TFontosInfo rekord három részből áll: a Név, a Fizetés és a TársAz deklarációból. Ezeket hívjuk mezőknek. A rekordtípus definiálásával lehető válik, hogy a program későbbi részében bármelyik változót deklaráljuk FontosInfo típusként. A TFontosInfo éppen olyan változó, mint a Delphi akármelyik beépített – pl. Integer vagy String – típusú változója. Természetesen ugyanúgy, mint más változókat, a TFontosInfo típusú változókat is deklarálni kell, mielőtt a programban használnánk azokat:

```
var
  FiktívAlk: TFontosInfo;
```

A rekord különböző mezőire a rekord nevével és a mezőazonosítóval hivatkozhatunk, a kettőt egymástól pont választja el.

```
FiktívAlk.Név := 'Kovács'
FiktívAlk.Fizetés := 100000;
FiktívAlk.TársAz := '036-78-9987';
```

A **with** kulcsszót – amellyel már foglalkoztunk a „Tulajdonságok beállítása" alfejezetben – rekordok mezőire történő hivatkozáskor is használhatjuk. Az alábbi példa ezt illusztrálja:

```
with FitívAlk do
  begin
    Név := 'Kovács';
    Fizetés := 100000;
    TársAz := '036-78-9987';
  end;
```

Rekord típusú változókból készíthetünk tömböt is:

```
Alkalmazottak: array [1..100] of FontosInfo;
```

Más változókhoz hasonlóan a rekordokat is deklarálhatjuk típusos konstansként. Inicializáláskor az egyes adatokat az alábbi példában látható módon választjuk el egymástól:

```
const
  Főnök: TFontosInfo = (Név: 'Fejes';
  Fizetés: 1000000; TársAz: '001-01-0011');
```

Ismét abból indulva ki, hogy a rekordtípusokat a Delphi a többi típushoz hasonlóan kezeli, a rekord mezői maguk is lehetnek rekordok. Például létrehozhatunk egy TFizetésiRekordot, amelyben az előző évi fizetés mellett a havi fizetések alakulását követhetjük nyomon:

```
type
  TFizetésiRekord = record
    JanFiz: Integer;
    FebrFiz: Integer;
    MárcFiz: Integer;
    ÁprFiz: Integer;
    MájFiz: Integer;
    JúnFiz: Integer;
    JúlFiz: Integer;
    AugFiz: Integer;
    SzeptFiz: Integer;
    OktFiz: Integer;
    NovFiz: Integer;
    DecFiz: Integer;
    ElőzőÉviFiz: LongInt;
end;
```

A TFontosInfo rekord így módosul:

```
type
  TKibővítettFontosInfo = record
  Név: String;
  Fizetés: TFizetésiRekord;
  TársAz: String[11];
end;
```

Így már egyetlen rekord adatainak megadása is sokkal nehezebb, ebben az esetben még nagyobb hasznát vesszük a **with** kulcsszónak. Az olyan rekordokban, mint a fenti TKibővítettFontosInfo, az egyes adatokra történő hivatkozás is bonyolultabbá válik. Minél több szintből áll egy rekord, ez annál inkább így van. A szinteket a szintaxisban pontok jelölik.

```
var
  MásikFiktívAlk: TKibővítettFontosInfo;
```

Ezzel egy új típusú változót hoztunk létre. Az egyik beágyazott mező értékét így írathatjuk ki:

```
ShowMessage (IntToStr(MásikFiktívAlk.Fizetés.
ElőzőÉviFiz));
```

Pascal tipp: *A Delphi nem támogatja a változó rekordokat, ehelyett használjuk ki az objektum öröklődését (l. 7. fejezet).*

■ *Mutatók*

A mutatókat (pointer) általában a legnehezebben emészthető típuskategóriának tartják. A következő pontok fejben tartásával megértésük jóval könnyebbé válhat:

- A program futásához szükséges minden információ a memória valamely részében tárolódik.
- Egy változó vagy más struktúra mutatója, annak memóriacímét tartalmazza.

Mint minden Object Pascal típust, a mutatókat is deklarálnunk kell, mielőtt a programban felhasználhatnánk azokat. A mutatók jelölésére a szintaxisban a ^ jelet használjuk. Általában a Delphibeli mutatók valamilyen előre kijelölt típusra mutatnak. Például:

```
EgészMutató: ^Integer;
ValósMutató: ^Double;
```

A mutatót egy változó memóriacímére az @ operátor vagy az ezzel egyenértékű Addr függvény segítségével lehet irányítani.

```
var
  Darab: Integer;
  DarabMutató: ^Integer;
begin
  Darab := 2;
  DarabMutató := @Darab {vagy addr (Darab)};
```

Mivel a Darab egy változó, értékét valahol tárolni kell. A fenti néhány sor végrehajtása után a DarabMutató tartalmazza ezt a címet. A DarabMutató értéke nem 2, hanem az a cím, ahol a Darab aktuális értéke található a memóriában.

Megjegyzés: A mutatókra vonatkozó konvenció az, hogy nagy P betűt (az angol pointer=mutató szóból) használunk azonosítójuk első karaktereként. Így a legtöbb program az előző DarabMutatót egyszerűen csak PDarab-ként használta volna.

Az inicializálatlan vagy hanyagul kezelt mutatók rendkívül veszélyesek. Használatuk alkalmazásunk látványos lefagyásához vezethet. Ennek oka abban keresendő, hogy a mutatók közvetlenül a memóriát kezelik, így ha figyelmetlenségből védett memóriaterületre írunk, akkor jobb esetben kapunk egy General Protection Fault (általános védelmi hiba) vagy Access Violation (védelemmegsértés) hibaüzenetet, attól függően, hogy a Delphi 32-t vagy a Delphi 16-ot használjuk, rosszabb esetben pedig a rendszert újra kell indítani. Ezért inicializáláskor a mutatóknak egy speciális értéket a **Nil**-t adjuk. (A **Nil** használatával jelezzük, hogy a mutató jelenleg nem

mutat semmilyen speciális memóriaterületre.) Ezért a mutatók használatakor mindig ellenőrizzük, hogy a mutató értékük **Nil**-e! (A Nil értékű mutatók használata önmagában kevés a memóriavédelmi hibák kikerülésére, az értékellenőrzés viszont elég!)

```
if PEgész = Nil then
  {Ne csináljunk semmit}
else
  {csinálhatunk bármit}
```

Pascal tipp: *A Delphi támogatja a Ptr függvényt, amellyel abszolút memóriacímeket tudunk kezelni. Abszolút címek használata Windows alatt azonban nem túl jó ötlet.*

Tegyük fel, hogy létrehoztunk egy mutatót, amely egy értelmes memóriacímre mutat. Ha erre a címre szeretnénk valamilyen adatot elhelyezni, akkor szükségünk van a ^ operátorra. Például:

```
PDarab^ := Darab + 4;
```

De a

```
Darab := PDarab + 4;
```

sornak nincs értelme, mert a Delphiben nem adhatunk mutatót egészhez (nem úgy, mint a C-ben vagy C++-ban).

A mutatók mutathatnak tömbökre és rekordokra is. Például:

```
type
  EgészTömb: array: [1..100] of Integer;
  PEgészTömb = ^ EgészTömb
var
  PéldaTömb: EgészTömb;
  TömbMutató: PEgészTömb;
```

Ilyen módon előállítottunk egy egészekből álló tömböt és egy mutatót, amely egy ilyen tömbre mutat. Az @ használatával a mutatót beállíthatjuk:

```
TömbMutató := @PéldaTömb;
TömbMutató^[99] := 37; {a 99. elem 37 lesz};
```

Annak köszönhetően, hogy a mutatókkal tömbökre és rekordokra is mutathatunk, lehetővé válik olyan bonyolult adatszerkezetek, mint láncolt listák és fák egyszerű létrehozása, valamint kezelhetőbbé válnak nagy adatstruktúrák paraméterátadásai is. (Az olvasó számára ismeretlen fogalmak megtalálhatók szinte bármelyik adatszerkezetekről szóló kiadványban.) Az alábbi példában egy egyszerű láncolt lista típusdeklarációját láthatjuk:

```
type
  PKövetkListaElem = ^ListaElem; {előzetes deklaráció}
  ListaElem = record
    ElsőRész: String;
    MásodikRész: String;
    KövetkezőRekord: PKövetkezőListaElem;
  end;
```

Az alapgondolat itt természetesen az, hogy a KövetkezőRekord mezőben tartalmazott mutató megadja a következő csatolt elem címét.

Megjegyzés: *Ez az egyedüli olyan eset, amikor hivatkozhatunk egy változóra, annak deklarálása előtt.*

Pascal tipp: *A Delphi, hasonlóan a Turbo Pascal újabb változataihoz, támogatja azon mutatók használatát, amelyek függvényekre és eljárásokra mutatnak. Ezeket igen ritkán használjuk. Amennyiben mégis abban a szokatlan szituációban találjuk magunkat, hogy egy ilyen mutatóra lenne szükségünk, akkor forduljunk a beépített help-hez. (Jó ha tudjuk, hogy minden eseménykezelő – többek között az OnClick is – valójában egy eljárásra irányított mutató.)*

Mutatók és a memóriakezelés

Ezidáig a mutatókat már meglévő objektumok memóriacímének tárolására használtuk. A mutatók igazi haszna az, hogy segítségükkel memóriát foglalhatunk le a program futása közben. Ennek legegyszerűbb módja a New eljárás használata. A következő példán tanulmányozhatjuk az eljárás működését:

```
var
  PEgészMutató: ^Integer;
begin
  New(PEgészMutató);
  PEgészMutató^ := 37;
end;
```

A New eljárás segítségével a mutató típusának megfelelő adat, jelen esetben egy egész számára foglaltunk le területet. Ekkor a

PEgészMutató^ := 37;

sor tölti meg a lefoglalt memóriát a megadott (37) értékkel. A New eljárással lefoglalt memóriát a Delphi egészen addig fenntartja, amíg nem használjuk a Dispose utasítást. (Általában ezt egy try/finally blokk belsejében tesszük a 8. fejezetben leírtaknak megfelelően.) A Dispose eljárás szintaxisa így néz ki:

Dispose(Mutató); {A mutatónak Nil értéket ad}

Megjegyzés: *Rendkívül fontos a New és Dispose eljárások összehangolása. Abban az esetben például, ha ugyanannak a változónak másodszorra is foglalunk memóriát anélkül, hogy először felszabadítanánk az első New által lefoglaltat, akkor a továbbiakban nem leszünk képesek az először lefoglalt területet használni.*

Pascal tipp: *A Delphi a Mark/Release eljáráspárt is támogatja memória lefoglalására, ill. felszabadításra.*

Minden esetben, amikor a New eljárást használjuk, a Delphi a memória egy speciális részében foglal memóriát a mutatónk számára. Ezt a területet globális heap-nek nevezik. Egy objektum számára foglalható maximális memória a Delphi 16-ban 64K, míg a Delphi 32-ben lényegében korlátlan (2 gigabyte), de a globális heap méretének (és így közvetett módon a mutatók által lefoglalható területnek) a Delphi mindkét verziójában korlátot szab a Windows. Sajnálatos módon a Delphi 16-ban van egy másik megkötés is: sem a statikus adattertület (globális heap – ahol a globális változókat és a típusos konstansokat tárolja a Delphi) által felhasznált memória, sem a lokális heap (ideiglenes tár pl. szerkesztést lehetővé tevő vezérlő elemek pufferje számára) mérete nem haladhatja meg a 64K-ot, azaz egy adatszegmens méretét. Ez magyarázza azt is, hogy miért használ olyan sok programozó a Delphi 16-ban ahol lehet, mutatókat: el szeretnék kerülni az adatszegmens megtelését.

A Delphi 16-ban lehetőség van a verem és a lokális heap méretének növelésére, amíg azok együttesen nem haladják meg a 64K-ot. A beállítást a Project Options dialógusablak Linker oldalán tehetjük meg. Ha a Delphi 16 alapértelmezésben beállított adattárolásra szolgáló memóriaméretei (lokális heap, globális heap, verem) nem bizonyulnak elegendőnek, akkor forduljunk a beépített help-hez, ott megtudhatjuk, hogyan kezeli a Delphi a heap-et.

Tipp: *Mivel a globális heap maximális méretét sok minden befolyásolja, célszerű ellenőrizni, hogy elegendő összefüggő memória áll-e a rendelkezésünkre, mielőtt a kívánt területet lefoglalnánk. Ezt a MaxAvail függvény segítségével tehetjük meg.*

Memóriablokkok lefoglalása és felszabadítása

A New és Dipose eljárásokat akkor használjuk, amikor egy objektum számára kívánunk helyet foglalni. Ha egy olyan objektum (pl. egy tömb) részére szeretnénk memóriablokkokat foglalni (majd felszabadítani), amelynek mérete változhat a projekt futása során, akkor a GetMem és FreeMem eljáráspárt használjuk. (A Dispose eljárás használatához hasonlóan a memóriafelszabadítást a **finally** blokkban végezzük – l. 8. fejezet.) Most a GetMem használatának illusztrálására egy nagyon egyszerű példát közlünk. (L. a későbbi megjegyzés rovatot is, ahol a GetMem he-

lyes használatát mutatjuk be, aminek megértéséhez a 8. fejezetben tárgyalt fogalmakra van szükségünk.)

```
type
  NagySztring = array [0..20000] of Char;
var
  NagySztringMutató: ^NagySztring;    {Használhattunk
  volna PChar-t is természetesen}
begin
  GetMem(NagySztringMutató, 20001);  {a globális
  heap-ben lefoglalt memóriaméret};
end;
```

Megjegyzés: *Fontos tudni, hogy az előző példában látott memóriafoglalást és -felszabadítást helyesen így valósíthatjuk meg.*

```
try
  NagySztringMutató^[0] := 'A';
  ...
  finally
    FreeMem(NagySztringMutató, 20001);
```

Ez a kódrészlet a 8. fejezetben bemutatandó try/finally struktúrát használja.

Visual Basic tipp: *A RedimPreserve Delphibeli megfelelőjeként a GetMem eljárást használhatjuk egy dinamikusan változó, tömbszerű struktúra felépítésére. (Egy másik lehetőség a TList objektum használata.)*

Nulla-végű sztringek és a PChar

Megjegyzés: Ez a fejezetrész elsősorban a Delphi 16 programozóknak nyújt segítséget. A Delphi 32 programozók minden gondja egy csapásra megoldódik a nagy sztring típus használnálatával (és a beépített memóriafelszabadítással az ún. hulladékgyűjtéssel).

Egy szokásos sztring a Delphi 16-ban nem más, mint egy karaktersorozat. A sorozat első byte-ját azonban a Delphi arra használja, hogy abban a sztringben található karakterek számát tárolja. Az a tény, hogy 1 byte-on csak 256 különböző számot tárolhatunk, egy szokásos sztring hosszát 255 karakterre korlátozza.

A Delphi mindkét verziója (a C nyomdokait követve) rendelkezik egy olyan típussal, amellyel ezt a korlátozást feloldhatjuk. A nulla-végű sztringekben akár 65 535 karaktert is tárolhatunk a Delphi 16-ban is. (A Delphi 32-ben lényegében nincs ilyen korlátozás – az alapbeállítás vagy nagy sztringek deklarálása ezt eleve biztosítja számunkra.)

Egy nulla-végű sztring a Delphi 16-ban nem más, mint egy olyan karaktersorozat, amely egy ASCII 0 (#) karakterre végződik. Ez a karakter jelzi a sztring végét. Ennek alapján mondhatjuk, hogy lényegében semmi különbség nincsen egy egy nulla-végű sztring és egy olyan „array[0 . . KarakterekSzáma] of Char" tömb között, amelynek utolsó eleme egy ASCII 0 karakter.

A Delphiben található előre definiált PChar típus segítségével elkerülhető a nulla-végű sztringek használatával járó sok vesződség, amit elsősorban a 0 kezdőindexű tömbökre irányított mutatók kezelése jelentene. A PChar-t fogjuk fel úgy, mint egy olyan mutatót, amely egy nulla-végű sztringre vagy egy azt reprezentáló tömbre mutat. Valójában a mutató a nulla-végű sztring első karakterére mutat. A Delphi az ASCII 0 karakter megkeresésével találja meg a sztring végét.

Mivel a PChar változók mutatók, ezért számukra memóriát kell foglalni (és utána természetesen felszabadítani) a globális heap-ben. Habár erre használhatnánk a GetMem/FreeMem eljáráspárt, mégis jóval gyakrabban a StrNew és StrDispose párt használjuk. A StrNew eljárásnak az az előnye, hogy használatával nem csak memóriát szabadítunk fel, hanem a sztringnek értéket is adunk egyben – e két fontos eljárással kapcsolatban további részleteket a beépített súgóból tudhatunk meg. Ezek a függvények különösen kényelmesek, amikor Object Pascal sztringeket kívánunk átadni Windows API függvényeknek, amelyek nulla-végű sztringeket vagy PChar típusú értékeket várnak.

Ha a bővített Delphi szintaxis (Extended syntax) direktíva be van kapcsolva, akkor a PChar-t úgy kezelhetjük, mint egy közönséges sztringet vagy tömböt. A direktívát a Project Options dialógusablak Compiler oldalán állíthatjuk be.

```
var
KarMutató: PChar;
KarTömb: array[0..1000] of Char;     {ez az adatszeg-
mensben tárolódik!}
begin
  KarMutató := KarTömb;     {A KarMutató a KarTömb
  memóriacímére mutat!};
  StrPCopy(KarMutató, 'Egy sztringet adok értékül
  egy PChar-nak');
  ShowMessage ('A nyolcadik karakter:' +
  Karmutató[7]);
end;
```

A StrNew függvényen kívül sok más olyan függvény van, amelyet hasznosnak találhatunk PChar változók használatakor. Például a StrPCopy segítségével közönséges (max. 255 karakter hosszú) sztringet másolhatunk nulla-végű sztringbe. A StrPas függvény pedig egy (kis méretű) nulla-végű sztringet konvertál közönséges sztringgé. PChar-ral kapcsolatos többi függvényről a beépített súgó „String Handling Routines (Null-terminated)" pontjában kaphatunk információkat.

Megjegyzés: *Megengedett, hogy egy PChar-nak közönséges sztringet adjunk értékül, amikor a bővített szintaxis opció be van kapcsolva, mégis célszerű elkerülni, mert ez az adatszegmens szerkezetének megbolygatását eredményezheti; jobban járunk a StrNew használatával.*

■ *Függvények és eljárások*

A függvény és az eljárás közti különbség abban áll, hogy a függvény egy értéket ad vissza, amíg az eljárás csak egyszerűen végrehajtódik, mint egy utasítás.

Pascal Tipp: *A Delphiben a függvények és eljárások közti különbségek sokkal árnyaltabbak lettek, mint amilyenek a „standard" Pascalban voltak. Az eljárás egyszerűen végrehajtódik, amikor neve különálló*

utasításként megjelenik. A függvényt általában egy kifejezésen keresztül hívjuk, ekkor a visszatérési érték szükséges a kifejezés kiértékeléséhez. A függvényt azonban az eljáráshoz hasonlóan is meghívhatjuk, ekkor a Delphi a visszatérési értéket nem veszi figyelembe (lényeges, hogy a bővített szintaxis opció be legyen kapcsolva).

Miként azt az előző fejezetben láthattuk, a külső adatokat, amelyekkel az eljárás vagy függvény dolgozik *paramétereknek* vagy *argumentumoknak* nevezzük. Vizsgáljuk meg például, hogy milyen paraméterekkel rendelkezik az előző fejezetben tárgyalt InputQuery függvény. A beépített súgóban az alábbi szintaxist találjuk a neve alatt:

```
function InputQuery(const ACaption, APrompt: String;
var Value: String): Boolean;
```

Látható, hogy a paraméter nevét egy kettőspont és a típusának megnevezése követi. Azt is könnyen észrevehetjük, hogy a közös típusú paraméterek típusát elég egyszer feltüntetni, ekkor a paramétereket egymástól vesszővel elválasztva soroljuk fel. A paraméterdeklarációkat egymástól pontosvesszővel választjuk el. A példafüggvénynek három paramétere van (ACaption, APrompt, Value), amelyek mindegyike sztring típusú.

Az ACaption és az APrompt a függvény úgynevezett konstans paraméterei (státuszukat a **const** kulcsszó jelzi). A konstans paraméterek cím szerint átadott paraméterek, de értéküket a függvény vagy eljárás nem változtatja meg. Ha az eljárásnak nem csak átadni akarunk értékeket, hanem azokat megváltoztatva vissza akarjuk kapni, vagy éppen az eljárásban kívánunk bizonyos értékeket kiszámítani, akkor az ún. változó paramétert használjuk. A változó paraméterek cím szerint adódnak át, értékük az eljárás során megváltozhat, ha egy értékadó utasítás bal oldalán állnak. A paraméterlistában ezeket a paramétereket a **var** kulcsszóval jelöljük meg.

Megjegyzés: *A függvényekben és eljárásokban szereplő paraméterneveknek nincs különösebb szerepük. A Delphi változónevekre vonatkozó konvenciónak ugyan meg kell felelniük, de a függvény hívásakor más nevű változók is szerepelhetnek paraméterként.*

Eseménykezelő eljárásokban egyaránt használhatunk konstans és változó paramétereket, Például egy TEdit komponens OnKeyPress eseménykezelőjének fejrészében az alábbiakat találjuk:

```
procedure TForm1.Edit1KeyPress(Sender: TObject;
var Key: Char)
```

Amikor a felhasználó lenyom egy billentyűt, akkor annak ASCII kódja a Key paraméter értékeként adódik át. Mivel ez egy változó paraméter, az eljárás belsejében értékét megváltoztathatjuk, és visszaküldhetjük a hívó TEdit komponensnek. Például ha egy ilyen programsor található az eljárásban:

```
If Key = '!' Then Key = #0;
```

akkor a felhasználó által beírt felkiáltójeleket a program kitörli.

Megjegyzés: Két másik billentyűzetet figyelő eseménykezelő eljárás is található a Delphiben. Ezek segítségével kezelhetjük azokat a vezérlőbillentyűket és a billentyűkombinációkat, amelyeknek nincs szabvány ASCII kódjuk. Olvassuk el a beépített helpben az OnKeyDown és az OnKeyUp-ról írottakat.

***Pascal tipp:** Kompatibilitási okok miatt, a Delphi támogatja a régebbi **value** paramétereket is. Ezek olyan paraméterek, amelyek érték szerint adódnak át, azaz a fordító másolatot készít a változóról, ami idő- és memóriaigényes. Sebesség szempontjából a **const** paraméter és a **var** paraméter között nincs különbség, a **const** paraméternek viszont kisebb a memóriaigénye. A beépített függvények és eljárások a konstans paramétereket használják a régi értékparaméterek helyett.*

A változó paraméterek cím szerint adódnak át (azaz a fordító a változó memóriacímét adja át). Ugyanez történik a **const** paraméterek esetén is, csak itt még egy további ellenőrzésre is szükség van: a fordító megvizsgálja, hogy megváltoztattuk-e a **const** paraméter értékét. Ha igen, akkor hibaüzenetet küld. Konstans paraméterek használatával kikerülhetjük azt az időveszteséget, amit eddig az okozott, hogy a függvénynek vagy eljá-

rásnak átadott **value** paraméterről a fordító lokális másolatot készített a program vermében.

Megjegyzés: *Eseménykezelő eljárások esetében a paraméterek Delphi objektumok is lehetnek. Lehetőség van az eseménykezelők direkt hívására is az eseményeken keresztüliek mellett (ilyen például egy magától lefutó demonstrációs program). A 7. fejezetben mindkettőről szólunk majd.*

A beépített Runtime Library (RTL)

A 6-3. táblázatban azoknak a gyakori függvényeknek a rövid leírását találjuk, amelyek a könyvben még nem szerepeltek. (A fájlkezelő rutinokat a 10. fejezetben tárgyaljuk.) Mivel többszáz beépített függvény áll rendelkezésünkre a Runtime Library-ban (futási idejű könyvtár), további információk szerzésére érdemes a beépített súgót böngészni. (Lehetőleg kezdjük a „Procedures and Functions Categorical List" ponttal.)

6-3. táblázat. A leggyakoribb beépített RTL rutinok

RTL rutin	*Leírás*
Abs	Az argumentum abszolútértékét adja vissza
AnsiCompareStr	Sztringösszehasonlítás, nagy- és kisbetűk különbözőek
AnsiCompareText	Szrtingösszehasonlítás, kis- és nagybetűk nem számítanak különbözőnek
AnsiLowerCase	A sztringet kisbetűssé alakítja
AnsiUpperCase	A sztringet nagybetűssé alakítja
AppendStr	Két sztringet egymáshoz kapcsol
ArcTan	Az argumentum arkusztangensét adja vissza radiánban.
Chr	A megadott ASCII/ANSII kódhoz tartozó karaktert adja vissza
CompareStr	Két sztring összehasonlítását végzi, nagy- és kisbetűk különbözőek
CompareText	Két sztring összehasonlítását végzi kis- és nagybetűk nem számítanak különbözőek
Concat	Sztringsorozatot fűz össze
Copy	Másolatot csinál a sztring egy szakaszáról
Cos	A radiánban megadott argumentum koszinuszát adja vissza
Date	A rendszer órájában tárolt dátumot adja vissza
DateTimeToStr	Egy idő formátumú értéket sztringgé alakít

6-3. táblázat. A leggyakoribb beépített RTL rutinok (folytatás)

RTL rutin	*Leírás*
DateToStr	Egy dátum formátumú értéket sztringgé alakít
DayOfWeek	A hét aktuális napját adja vissza
Delete	Az argumentumként megadott sztring egy részét törli
Exp	Az e számot (a természetes alapú logaritmus alapját) az argumentumban megadott kitevőre emeli
Frac	Az argument törtrészét képezi
Insert	Egy sztringet szúr be egy másik sztringbe
Int	Az argument egészrészét képezi, valós értéket ad vissza
IntToHex	Egy decimális alakban megadott egész szám hexadecimális alakját adja vissza
IsValidIdent	Megvizsgálja, hogy a sztring egy szabályos Delphi azonosító-e
Length	A megadott sztring hosszát adja vissza
Ln	Az argumentum természetes alapú logaritmusát képzi
LowerCase	A sztringet kisbetűssé alakítja
Now	Megadja a rendszerórában tárolt aktuális dátumot és időt
Pi	14 tizedesjegy pontossággal megadja a π értékét
Pos	Egy sztring helyét keresi meg egy másik sztringben
Random	Egy (ál) véletlenszámot generál
Randomize	A rendszeridő alapján inicializálja a véletlenszám-generátort
Round	Egy lebegőpontos szám egészrészét adja meg egész számként
Sin	A radiánban megadott argumentum szinuszát adja vissza
Sqr	Az argumentum négyzetét számítja ki egy gyors algoritmus segítségével
Sqrt	A lehető leggyorsabb módon számolja ki az argumentum gyökét
StrPas	Nulla-végű sztringet alakít Pascal-stílusú sztringgé; csak akkor működik a Delphi 16-ban, ha a nulla-végű sztring 256 karakternél rövidebb
StrPCopy	Egy Pascal-stílusú sztringet alakít nulla végű sztringgé
StrToDate	Egy sztringet dátum formátumúvá alakít
StrToDateTime	Egy sztringet dátum/idő formátumúvá alakít
StrToTime	Egy sztringet idő formátumúvá alakít
Time	Az aktuális időt adja meg a rendszeróra alapján
Trunc	A lebegőpontosszám tizedespont előtti értékét adja vissza egész számként
UpCase	A megadott karaktert nagybetűssé alakítja
UpperCase	A megadott sztringet nagybetűssé alakítja

Megjegyzés: A Delphi 32 RTL-jának sztringkezelő rutinjai mind hosszú sztringekkel dolgoznak. A Delphi 16-ban ezek csak a max. 255 karakterből álló sztringeket fogadják el. A Delphiben külön függvények találhatók a null-végű sztringek (PChar típus) kezelésére. Ezeknek a beépített help-ben nézhetünk utána – előfordulhat, hogy a függvények kezelése mutató-aritmetikát igényel.

Megjegyzés: Miként az a 6-3. táblázatban is látható, a beépített RTL függvények között sok olyan van, amelyek bemenő paramétere más típusú, mint a visszatérési értéke. Ezek a típuskonverziós függvények rendkívül értékesek az olyan szoros típusmegkötésű nyelvekben, mint a Delphi.

A Windows API használata

A Windows API (Application Programming Interface) függvényei szigorúan véve nem a Delphi beépített függvényei, a Delphi csak asszisztál a Windows-t alkotó többszáz speciális függvény hívásához. Ezeket Windows API vagy egyszerűen API függvényeknek nevezzük.

Figyelem: Ha nem kellő elővigyázatossággal használjuk az API függvényeket, akkor rendszerünk összeomolhat, amin csak a gép újraindítása segít. Mielőtt elkezdenénk kísérletezgetni a Windows API hívásokkal, ajánlatos az automatikus mentést lehetővé tenni az Environment Options dialógusablak Preferences oldalán.

A Delphi lehetőségekben (beépített függvények, eljárások) olyan gazdag, hogy a legtöbb programozási feladatot meg lehet valósítani a Windows API függvények igénybevétele nélkül. Néhány speciális esetben azonban mégis ezek használatára kényszerülünk. Ha meg szeretnénk tudni, hogy a felhasználó számítógépe milyen processzorral működik (30 386, 486, Pentium), van-e matematikai koprocesszora (80 387 vagy 486DX vagy Pentium), a felhasználó balkezes egérbeállítást használ-e, akkor a megfelő API függvényt kell használnunk. Még olyan függvények hívására is van lehetőségünk, mint az ExitWindows, amelynek meghívása a Windows bezárását eredményezi. (A Windows API függvények széles körű használatához szükséges a téma mélyreható tanulmányozása. Készüljünk fel arra, hogy ez nem kevés idő: a standard hívások

dokumentációja kb. 2000 oldalt tesz ki. A 16-bites és a 32 bites API között kisebb-nagyobb különbségek vannak, amelyekre szintén jó, ha odafigyelünk.)

Tipp: A Delphi mindkét verziója tartalmazza a megfelelő Windows API hivatkozásokat, amelyek a Delphi szintaxisához igazodnak (és nem a C-hez). Az API függvények fejrészét szabadon másolhatjuk a programon belül.

A Delphi 16 API egyik legegyszerűbb és egyben nagyon hasznos függvénye, a Longint értékkel visszatérő GetWinFlags. (A Win 32 ehhez legközelebb álló függvénye a GetSystemInfo.) A visszakapott egész szám segítségével meghatározhatjuk a számítógép processzorának típusát. Az alábbi Delphi 16 alá írt példa ezt a függvényt használja (a Delphi 32-ben ez nem fut).

```
var
  VizsgálandóEgész: Longint;
begin
  VizsgálandóEgész := GetWinFlags;
    if (VizsgálandóEgész and WF_80x87) = WF_80x87
    then
      ShowMessage ('A gépben van matematikai
      koprocesszor');
    if (VizsgálandóEgész and WF_CPU286) = WF_CPU286
    then
      ShowMessage ('Még mindig 286-os processzort
      használ?');
```

Egy másik fontos példa, amely szintén rámutat az API függvények hasznára, a GetSystemMetrics függvény, amelynek szintaxisa így néz ki:

function GetSystemMetrics(Index: Integer): Integer;

Paraméterként egy egész értéket adunk át, amely jelzi a függvénynek, hogy milyen információra vagyunk kíváncsiak. (A visszaadott értékek megtalálhatók a Delphi előre definiált konstansai között. Ezeket a beépí-

tett sugó API-ról szóló részében megtalálhatjuk.) A GetSystemMetrics függvény nagyon részletes információkkal szolgál a rendszerről, amelyben az alkalmazás fut. Többek között ennek a függvénynek a használatával tudhatjuk meg a Windows-tól, hogy csatlakozik-e egér a számítógéphez, és ha igen, akkor az jobb- vagy balkezes beállítású-e. A közölt programrész az előbbi ellenőrzést valósítja meg:

```
if GetSystemMetrics (SM_MOUSEPRESENT)<> 0 Then
  ShowMessage ('Önnek van egere.') ;
```

Megjegyzés: *A Windows API hívásoknál gyakran szükséges a hívó objektum Windows-azonosítójának (handle) elküldése. A Delphiben ez az ún. Handle (futás idejű) tulajdonság értéke. Handle tulajdonság minden ablakkal rendelkező komponenshez tartozik.*

■ *Felhasználó által definiált függvények és eljárások*

Mint minden magasabb rendű programnyelvben, a Delphiben is lehetőség van a program kezelhető elemekre bontására. A Visual Basic-hez hasonlóan (a C-vel ellentétben) a Delphi megkülönbözteti a függvényeket az eljárásoktól, az előbbiek valamilyen értéket adnak vissza, míg az utóbbiak „üres kézzel" térnek vissza. A Delphiben a függvényeket és eljárásokat közös néven alprogramoknak nevezzük. A kifejezés segít észben tartani, hogy valójában minden függvény és eljárás egy önálló mini-programként viselkedik. Többek között lehetnek saját típusaik, konstansaik, típusos konstansaik és változóik, amelyeket azokban deklarálunk ill. definiálunk. Sőt további alprogramokat is tartalmazhatnak. (Miért érdemes egy alprogramot egy másik alprogramba beágyazni? A választ az alprogramok hatáskörére vonatkozó szabály adja: egy beágyazott alprogram lokális a tartalmazó programra nézve, azaz a program más részeiből nem érhető el. Másrészről a beágyazott alprogramban deklarált változók sehol máshol nem láthatók.)

Megjegyzés: *Hogy milyen kis elemekre bontunk egy programot, az teljesen ízlés dolga. Én arra törekszem, hogy sem az eseménykezelő eljárások sem az alprogramok ne legyenek egy képernyőnél sokkal hosszabbak.*

Példaként mutatunk egy olyan programrészletet, amely egy karakter előfordulását számolja meg egy sztringben, és a kapott egész értékkel tér vissza:

```
function KarakterElőfordulás (const Szöveg: String;
const Karakter: Char): Integer;
var
  i, szám, SztringHossz: Integer;
begin
  SztringHossz := Length (Szöveg);
  szám := 0;
  for i := 1 to SztringHossz do
    begin
      if Szöveg[i] = Karakter then szám := szám + 1;
    end;
  KarakterElőfordulás := szám;
end;
```

Ezen a függvényen keresztül mutatjuk be a függvények általános felépítését. Először is vegyük szemügyre a *függvényfejet*:

```
function KarakterElőfordulás (const Szöveg: String;
const Karakter: Char): Integer;
```

Az alprogramokat mindig egy fejrész vezeti be. A fejrész első szava utal az alprogram fajtájára. A function szó a függvényt, a procedure az eljárást jelöli. Ezen kívül itt található az alprogram neve és a formális paraméterlista. Az alprogram nevének meg kell felelnie a Delphi azonosítókra vonatkozó szabályoknak (l. 4. fejezet). A paraméterek típusát fel kell tüntetni, ezt a kettőspont után tehetjük meg. A különböző típusú paramétereket pontosvessző választja el egymástól. Valójában a felhasználó által definiált függvények és eljárások szintaxisa megegyezik a beépített Delphi alprogramokéval. Például az általunk létrehozott alprogramokban lehetnek ugyanúgy **var**, **const** és value paraméterek. (A KarakterElőfordulás függvényünkben csak konstans paraméter fordul elő, hiszen nem kívánunk a paraméterek eredeti értékén változtatni.)

tett sugó API-ról szóló részében megtalálhatjuk.) A GetSystemMetrics függvény nagyon részletes információkkal szolgál a rendszerről, amelyben az alkalmazás fut. Többek között ennek a függvénynek a használatával tudhatjuk meg a Windows-tól, hogy csatlakozik-e egér a számítógéphez, és ha igen, akkor az jobb- vagy balkezes beállítású-e. A közölt programrész az előbbi ellenőrzést valósítja meg:

```
if GetSystemMetrics (SM_MOUSEPRESENT)<> 0 Then
  ShowMessage ('Önnek van egere.') ;
```

Megjegyzés: *A Windows API hívásoknál gyakran szükséges a hívó objektum Windows-azonosítójának (handle) elküldése. A Delphiben ez az ún. Handle (futás idejű) tulajdonság értéke. Handle tulajdonság minden ablakkal rendelkező komponenshez tartozik.*

■ *Felhasználó által definiált függvények és eljárások*

Mint minden magasabb rendű programnyelvben, a Delphiben is lehetőség van a program kezelhető elemekre bontására. A Visual Basic-hez hasonlóan (a C-vel ellentétben) a Delphi megkülönbözteti a függvényeket az eljárásoktól, az előbbiek valamilyen értéket adnak vissza, míg az utóbbiak „üres kézzel" térnek vissza. A Delphiben a függvényeket és eljárásokat közös néven alprogramoknak nevezzük. A kifejezés segít észben tartani, hogy valójában minden függvény és eljárás egy önálló mini-programként viselkedik. Többek között lehetnek saját típusaik, konstansaik, típusos konstansaik és változóik, amelyeket azokban deklarálunk ill. definiálunk. Sőt további alprogramokat is tartalmazhatnak. (Miért érdemes egy alprogramot egy másik alprogramba beágyazni? A választ az alprogramok hatáskörére vonatkozó szabály adja: egy beágyazott alprogram lokális a tartalmazó programra nézve, azaz a program más részeiből nem érhető el. Másrészről a beágyazott alprogramban deklarált változók sehol máshol nem láthatók.)

Megjegyzés: *Hogy milyen kis elemekre bontunk egy programot, az teljesen ízlés dolga. Én arra törekszem, hogy sem az eseménykezelő eljárások sem az alprogramok ne legyenek egy képernyőnél sokkal hosszabbak.*

Példaként mutatunk egy olyan programrészletet, amely egy karakter előfordulását számolja meg egy sztringben, és a kapott egész értékkel tér vissza:

```
function KarakterElőfordulás (const Szöveg: String;
const Karakter: Char): Integer;
var
  i, szám, SztringHossz: Integer;
begin
  SztringHossz := Length (Szöveg);
  szám := 0;
  for i := 1 to SztringHossz do
    begin
      if Szöveg[i] = Karakter then szám := szám + 1;
    end;
  KarakterElőfordulás := szám;
end;
```

Ezen a függvényen keresztül mutatjuk be a függvények általános felépítését. Először is vegyük szemügyre a *függvényfejet*:

```
function KarakterElőfordulás (const Szöveg: String;
const Karakter: Char): Integer;
```

Az alprogramokat mindig egy fejrész vezeti be. A fejrész első szava utal az alprogram fajtájára. A function szó a függvényt, a procedure az eljárást jelöli. Ezen kívül itt található az alprogram neve és a formális paraméterlista. Az alprogram nevének meg kell felelnie a Delphi azonosítókra vonatkozó szabályoknak (l. 4. fejezet). A paraméterek típusát fel kell tüntetni, ezt a kettőspont után tehetjük meg. A különböző típusú paramétereket pontosvessző választja el egymástól. Valójában a felhasználó által definiált függvények és eljárások szintaxisa megegyezik a beépített Delphi alprogramokéval. Például az általunk létrehozott alprogramokban lehetnek ugyanúgy **var**, **const** és value paraméterek. (A KarakterElőfordulás függvényünkben csak konstans paraméter fordul elő, hiszen nem kívánunk a paraméterek eredeti értékén változtatni.)

A függvények esetén szükséges a visszatérési érték típusát feltüntetni a fejrész végén. (A példánkban ez egy egész típus.) Végül a függvény visszatérési értékét egy olyan értékadással adhatjuk meg, amelynek a bal oldalán a függvény neve szerepel. A példaprogramban ez a

```
KarakterElőfordulás := szám;
```

sor volt. A függvényben a vissszatérési értéket az implicit módon definiált lokális Result változón keresztül is megadhatjuk. Például az alábbi sorral helyettesíthetnénk az előzőt:

```
Result := Szám;
```

A fenti példa alábbi, rövidebb változatában a Result használatát is megfigyelhetjük:

```
function KarakterElőfordulás (const Szöveg: String;
const Karakter: Char): Integer;
var
  Index : Integer;
begin
  Result := 0; {Ezt használjuk a függvénynév
  helyett};
  for Index := 1 to Length (Szöveg) do
  if Szöveg[Index] = Karakter then Inc(Result);
end;
```

Lássunk most példát egy eljárásra is. Írjunk egy olyan alprogramot, amelyik egy sztringet és két különböző karaktert kap meg paraméterként, és a sztringben minden olyan helyre, ahol az első karakter szerepel a másodikat helyezi:

```
procedure KarakterCsere(var Szöveg: String; const
RégiKarakter, ÚjKarakter: Char);
var
  i, SztringHossz: Integer;
begin
```

```
  SztringHossz := Length (Szöveg);
  for i := 1 to SztringHossz do
    begin
      if Szöveg[i] = RégiKarakter then Szöveg[i] :=
      ÚjKarakter;
    end;
end;
```

Ebben az esetben meg akarjuk változtatni az eredeti sztringet, ezért azt változó paraméterként adjuk át. A másik két paramétert az alprogram változatlanul hagyja, ezért elég azokat **const** paraméterként átadni.

Alprogramok elhelyezése

Az alprogramot mindig a unit implementációs blokkjába helyezzük el. Ezeket az alprogramokat használhatjuk (szakszóval *meghívhatjuk*) a unit bármilyen eseménykezelő eljárásából vagy alprogramjából. (Lehetőség van ennek a korlátozásnak a megkerülésére. Lásd a fejezet rekurzióról szóló részét.)

Emlékezzünk vissza, hogy a unit **uses** sorában a meghívható unitok vannak feltüntetve. Felmerülhet az a természetes igény, hogy egy adott unit (az egyszerűség kedvéért legyen Unit1) speciális alprogramját elérjük egy másik unit (legyen Unit2) alprogramjaiból, ekkor nem elég a megfelelő azonosítót beírni a Unit2 **uses** sorába, hanem a használni kívánt alprogram fejrészének másolatát is el kell helyeznünk az azt tartalmazó unit (ebben az esetben a Unit1) **interface** blokkjában.

Azon alprogramoknak, amelyek fejét feltüntetjük a unit **interface** részében, hatáskörük globális. Többek között az eseménykezelő eljárások fejei mindig megtalálhatóak az **interface** blokkban, így biztosítva azt, hogy más unitok számára is elérhetők legyenek, a hívó unit **uses** sorának kiegészítése után.

Alprogramok használata

Ha egy eljárást a saját unitjának egy részéből kívánunk meghívni, akkor egyszerűen adjuk meg az eljárás nevét és utána zárójelben a szükséges paramétereket. Létre szeretnénk például hozni egy olyan TButton kom-

ponenst, amely egy szerkesztési ablak (Edit1) szövegében minden „U" betűt „W"-re cserél. Kínálja magát a KarakterCsere eljárás használata, ehhez azonban azt el kell helyeznünk a form implementációs blokkjában. Az eseménykezelő maga így néz ki:

```
procedure TForm1.Button1Click (Sender: TObject);
var
  IdeiglSztring : String;
begin
  IdeiglSztring:= Edit1.Text;
  KarakterCsere (IdeiglSztring, 'U', 'W') ;
  Edit1.Text := IdeiglSztring;
end;
```

Ha egy függvényt hívunk, akkor azt rendszerint egy értékadáson vagy kifejezésen keresztül tesszük. Például:

```
procedure TForm1.Button1Click (Sender: TObject) ;
var
  IdeiglSztring: String;
begin
  If  KarakterElőfordulás (Edit1.Text, 'U') = 0 then
    ShowMessage ('A szövegben nincs ''U'' betü')
  else begin
    IdeiglSztring := Edit1.Text;
    KarakterCsere (IdeiglSztring, 'U', 'W');
    Edit1.Text := IdeiglSztring;
  end;
```

Megjegyzés: Ha a bővített szintaxisfordítási direktívát állítottuk be, akkor a függvényt eljárásként is meg lehet hívni, ekkor a visszatérési értéket nem vesszük figyelembe.

Amennyiben egy globális hatáskörű (az **interface** blokkban deklarált) alprogramot szeretnénk meghívni egy másik unitból, használjuk a unit nevét és egy pontot az alprogram neve előtt. Miután a Unit2-t felvettük

a Unit1 **uses** sorába, a Unit2 egy eljárását a Unit1-ből így hívhatjuk meg:

```
Unit2.EljárásNév (paraméterek);
```

Tömbök átadása alprogramoknak

A Delphi, a Turbo Pascal 7-hez hasonlóan, támogatja a *nyitott* tömbparaméterek használatát az alprogramokban. A nyitott tömbparaméter lehetőséget biztosít arra, hogy olyan alprogramokat alakítsunk ki, amelynek bármilyen méretű tömböt átadhatunk. Az alprogramban a tömb méretét így nem kell előre megadnunk. Egy nyitott paramétert használó alprogramban csak a tömb típusát kell megjelölni. Híváskor a paraméterlistában viszont az átadni kívánt tömb indextartományát is meg kell adni a szokásos szintaxis használatával:

TömbNév: array[KezdőIndex ... ZáróIndex] of Típus

Az alprogramon belül az aktuális tömb indextartományát a Low és High előre definiált függvényekkel határozhatjuk meg. A Low a kezdő indexet a High pedig a záró indexet adja meg. Nézzünk példaként egy olyan függvényt, amelyik egy egész tömb elemeinek az átlagát számolja ki:

```
function Átlag (EgészTömb: Array of Integer) :
Double;
var
  Elemszám, i: Integer;
  Összeg: Longint;
begin
  Összeg := 0;
  Elemszám := High (EgészTömb) - Low(EgészTömb) +1;
  For i := Low (EgészTömb) to High (EgészTömb) do
    Összeg := Összeg + EgészTömb[i];
  Átlag := Összeg/Elemszám;
end;
```

■ *Külső DLL-ek használata*

A Delphiben majdnem minden programot meg lehet írni anélkül, hogy más nyelvekben megírt függvényeket kéne meghívnunk. A leggyakoribb kivételek ez alól:

- A fejezet korábbi részében tárgyalt Windows API függvények alkalmankénti használata
- A C-ben vagy C++-ban írt könyvtári függvények

Fontos tudni, hogy a C/C++ függvények neveinél lényeges a kis- és nagybetűk megkülönböztetése (a Delphi függvényektől eltérően). Egy másik probléma abból adódik, hogy a szabványos C/C++ modellben a függvények a vermet fordított sorrendben olvassák ki. Könnyebb a dolgunk, ha a C/C++ kód úgy volt lefordítva, hogy a függvényfejben a PASCAL kulcsszó szerepelt. Nem éri meg (szinte reménytelen) olyan DLL-ek exportált (amikhez kívülről hozzá lehet férni) függvényeit használni, amelyeket úgy fordítottak le, hogy fejrészükben nem szerepelt ez a kulcsszó.

Azon DLL-ek, amelyek a fenti módon voltak lefordítva (és a legtöbb DLL ilyen), egyszerűen elérhetőek a Delphiből. Ahhoz, hogy a DLL-ek exportált függvényeit használhassuk, egyedül a nevüket szükséges ismerni. A dolog egyszerű, ha ismerjük a fordítót (és ahhoz van hozzáférési lehetőségünk), amelyben az adott DLL-t írták, vagy rendelkezünk egy a BoundChecker-hez hasonló segédeszközzel. Minden fordítóban van lehetőség a DLL-ek tartalmának elemzésére, így meghatározhatjuk exportált függvényei nevét. A Borland C++-ban az ezt megvalósító eszköz a TDUMP.EXE.

Tegyük fel, hogy van egy @VégtelenSzorzás nevű eljárásunk a NagyAritmetika könyvtárban (DLL-ben). Mivel a Delphi eljárásokban nem szerepelhet az „@" jel, ezért azt először át kell keresztelnünk. A Delphi 16-ban az alábbiakat illesszük be a unit implementációs blokkjába:

```
procedure VégtelenSzorzás; {Delphi 16}
  far; external 'NagyAritmetika' name @VégtelenSzorzás
```

A Delphi 32-ben:

```
procedure VégtelenSzorzás; {Delphi 32}
  external 'NagyAritmetika.dll' PASCAL name
  '@VégtelenSzorzás';
```

Ezek után a VégtelenSzorzás-t úgy használhatjuk, mint bármilyen más Delphi eljárást.

Megjegyzés: A Delphi 32-ben nem létezik a ***far*** *(távoli szegmensek közti) hívási modell fogalma. Ezért a kulcsszó nem szerepel a Delpi 32-re írt példában, viszont jelezni kell a C DLL-ek hívását, mert a Delphi 32 alapértelmezett hívási modellje a FastCall, amely a paramétereket nem a szokványos Pascal-modell szerint adja át.*

■ *Rekurzió*

Rekurziónak nevezik azt az általános módszert, amellyel úgy oldunk meg problémákat, hogy azokat hasonló, de egyszerűbb feladatokká alakítjuk. Az általános recept problémák rekurzív módon történő megoldására a következő:

OldjukMegAProblémátRekurzióval (probléma)
  Ha a probléma triviális, akkor egyszerű műveletetekkel határozzuk meg a megoldást
  egyszerűsítsük a problémát
OldjukMegAProblémátRekurzióval (egyszerűbb probléma)
  Az egyszerűsített probléma megoldásából határozzuk meg az eredeti megoldását

A rekurzív alprogram addig hívja újra meg újra önmagát, minden lépésben egyszerűsítve a problémát, amíg az át nem megy a triviális esetbe, ekkor az alprogram véget ér. A tapasztalt programozó számára a rekurzív gondolkodásmód egyes problémák esetén rendkívül elegáns megoldásokhoz és ennek következtében elegáns programokhoz vezet. Például a leg-

több nagyon gyors rendezés – többek között a QuickSort – is ezen az elven működik.

A Delphi programozók számára a rekurzió egy gyakori alkalmazási területét, a különböző rendezési rutinok mellett, a háttértárak könyvtárrendszerének kezelése jelenti. Például ha le szeretnénk törölni egy olyan állományt, amelynek helyét nem ismerjük, akkor egyre mélyebben található alkönyvtárakban kellene keresgélnünk a fájlrendszerben, amíg ki nem merítettük az összes alkönyvtárat a lemezen.

Két jól elkülöníthető típusát különböztethetjük meg a rekurziónak. Az elsőben az alprogram csak önmagát hívja meg. Ezt direkt rekurziónak nevezzük. A direkt rekurzió használata programozástechnikailag egyszerű: az alprogramot a szokásos (a már látott) módon kell mindig meghívnunk. A második típust indirekt rekurziónak hívják. Erről akkor beszélünk, amikor egy alprogram meghív egy másik alprogramot, amely azután meghívja az első alprogramot.

Nézzünk a rekurzióra egy gyakori példát: két egész szám legnagyobb közös osztójának (LNKO) meghatározását. (Azok számára, akik elfelejtették a középiskolában tanultakat: a legnagyobb közös osztó a két szám közös osztói közül a legnagyobb. Ezt használjuk például törtek összeadásánál is.)

LNKO (4,6) = 2 (mert 2 a legnagyobb egész szám, amely a 4-et és a 6-ot is osztja).
LNKO (12,7) = 1 (mivel egynél nagyobb szám nem osztja a 12-t és a 7-et egyszerre).

Körülbelül 2000 évvel ezelőtt Euklidész az alábbi algoritmust fedezte fel két szám LNKO-jának meghatározására:

Ha a osztója b-nek, akkor az LNKO (a,b) = b. Ha nem, akkor
LNKO (a,b) = LNKO (b, a mod b)

(Emlékezzünk, hogy a mod az osztási maradékot képező függvény)
Például:

LNKO(126, 12) = LNKO (12, 126 mod 12) = LNKO (12,6)= 6

A LNKO-t kiszámoló rekurzív függvény kódja így néz ki:

```
function LNKO (Const P, Q: Integer): Integer;
begin
  If Q mod P = 0 Then
    LNKO := P
  else
    LNKO := LNKO (Q, P mod Q);
end;
```

A függvény implementációjában először ellenőrizzük, hogy a triviális esettel van-e dolgunk. Ha nem, akkor a kód gondoskodik a probléma egyszerűsítéséről, hiszen a maradékképzés kisebb számokhoz vezet. (Ebben az esetben nincs szükség a részeredmények használatára, nem úgy, mint pl. egy rendezési rutinnál.)

Az indirekt rekurzióban a függvény vagy eljárás egy közvetítőn keresztül hívja meg saját magát. Például az A függvény meghívja a B függvényt, de a B is meghívja A-t. Ennek megvalósítását megnehezíti az a tény, hogy a Delphiben általában egy alprogram nem használhatja az utána definiált alprogramot.

Ezt a problémát egy speciális deklaráció használatával, az úgynevezett előre hivatkozással kerülhetjük meg. Az előre hivatkozást az alprogram fejének végén kell jelölni a **forward** kulcsszóval és egy pontosvesszővel. Ezután helyezzük a forward kulcsszóval kibővített fejrészt annak az alprogramnak a kódja elé, amelyik majd hívja.

A következő kódrészlet példa a **forward** használatára:

```
procedure Második (szám: Integer); forward;
procedure Első (darab: Integer);
begin
  Második(1);
end;
procedure Második (szám: Integer);
var
  I: Integer;
begin
  for I := 1 to szám do
    ShowMessage ('Működik');
end;
```

Megjegyzés: Nagyon bonyolult rekurzív programok esetén szükségünk lehet a Delphi 16 Verem méretének alapbeállítását megváltoztatni (16 384 byte). Ezt a Project Options dialógusablak Linker oldalán tehetjük meg. A Delphi 16 beállítható maximális veremmérete 65 536 byte. (A Delphi 32-ben ez 1 megabyte és szükség esetén növelhető.)

■ *Projektek és unitok kezelése*

Fejlesztés alatt a projektek tárolják az alkalmazással kapcsolatos összes információt. A View | Project Manager menüpont választásakor megjelenő dialógusablakon keresztül könnyen hozzáférhetünk a projekt unitjaihoz és formjaihoz. Ezen a dialógusablakon keresztül illeszthetünk be a projektbe, illetve törölhetünk onnan formokat és unitokat.

A projekttel kapcsolatos beállítások nagy részét a Project Options dialógusablakban végezhetjük. Például – miként az előző fejezetben láttuk – a Forms oldalt használjuk annak kijelölésére, hogy mely formokat hozza létre a Delphi automatikusan. Az Application oldalon pedig kiválaszthatjuk alkalmazásunk ikonját. (A Compiler-oldalról a „Lényeges tudnivalók a fordítóról" alfejezetben lesz szó. Információkat a dialógusablak azon részeiről, amelyeket a könyvben nem részletezünk, az olvasó a beépített súgóban találhat.)

A projektet egyetlen fájl (.DPR) írja le, amely tartalmazza a projekt összes formját és unitját. Habár lehetőségünk van a projekt fájl vizsgálatára és szerkesztésére is a View | ProjectSource pont választásával, általában jobb, ha a Delphire bízzuk ennek a fájlnak a karbantartását. Egyedül akkor szükséges hozzányúlnunk a projekt fájlhoz, ha egy önálló alkalmazás helyett egy DLL-t szeretnénk létrehozni vagy valamilyen indulási kódra (start-up code) van szükségünk, pl. egy hátteret szeretnénk megjeleníteni, mielőtt a főform megjelenik.

***Tipp:** Célszerű minden projektnek egy külön könyvtárat fenntartani, ahol a projekttel kapcsolatos összes fájlt tárolhatjuk. A Delphi alapbeállításban a \BIN alkönyvtárban tárolja az összes állományt, ami egy idő után kisebb zűrzavarhoz vezethet.*

Az egyszerűség kedvéért a Delphi ugyanazt a nevet adja a projekt fájlnak, mint a lefordított .EXE állománynak. Azt, hogy a Delphi melyik könyvtárba helyezze a fordításkor létrehozott .EXE fájlt, a Project Options dialógusablak Directories | Conditionals oldalán állíthatjuk be (az alapbeállítás a Delphi \BIN alkönyvtára).

Tipp: *Minden alkalommal, amikor programjainkat a Delphi fejlesztői környezetén belülről futtatjuk, a Delphi előállít egy .EXE fájlt, ezért időnként szükség lehet a \BIN alkönyvtárban felhalmozódott, használaton kívüli .EXE fájlok letörlésére – feltéve, hogy nem változtatunk ezen az opción.*

Bővebben a unitok részeiről

A projekt programkódját untiokban tároljuk, ezek adják a Delphiben a moduláris programozás alapját. Amikor hasznos rutinokat tartalmazó könyvtárakat szándékozunk gyártani, akkor ezeket valószínűleg unitokban tárolnánk. Az a lehetőség, hogy a projekteket kisebb egységekből (unit) építhetjük fel, egyszerűbbé teszi a tesztelt rutinokból álló könyvtár hozzáadását projektünkhöz. A külső kiegészítő segédeszközöket is általában unitok formájában árulják. Sokan úgy építik fel a programot, hogy az összes szubrutint külön unitokban tárolják, és azok függvényeinek és eljárásainak meghívása az eseménykezelő eljárásokban történik.

- Új unitot úgy adhatunk a projekthez a Delphi 32-ben, hogy a File menü File | New alpontját választjuk, majd a megjelenő dialógusablakban a New Unit ikonra kattintunk, és lenyomjuk az OK gombot.
- A Delphi 16-ban a File | New Unit választással adhatunk új unitot a projekthez.

Tipp: *Azokat a unitokat, amelyeket több projektben is használunk (vagy kívánunk majd használni), érdemes lehet egy külön alkönyvtárban tárolni. Ilyen esetben ne felejtsük el a Project Options dialógusablak Directories/Conditionals oldalán található Search Path ablakban megadni a könyvtár elérési útját.*

Minden alkalommal, amikor egy új formot (.DFM) illesztünk a projecthez, a Delphi automatikusan létrehoz egy ahhoz tartozó .PAS állományt (unitot). Minden unitnak lehet saját felhasználó által definiált típusa, globális és lokális változói, alprogram-deklarációi és így tovább. A unit megnyitásakor a Delphi feltünteti annak részeit:

- a unit fejét,
- az **interface** blokkot,
- az implementációs blokkot.

*Megjegyzés: A unit végét az **end** kulcsszó és egy pont jelzi.*

A negyedik részt **initialization** blokknak hívják, használata opcionális mindkét Delphi változatban. A Delphi 32-ben lehetőség van még egy **finalization** blokk hozzáadására, ebbe a program azon része kerül, amelyet az alkalmazás lezárása előtt szeretnénk még végrehajtani.

*Megjegyzés: A Delphi 32 **finalization** blokkja a Delphi 16 ExitProcjának felel meg.*

Ismerkedjünk meg a unitok ezen részeivel külön-külön is:

A unit feje

A unit fejét a **unit** kulcsszó jelöli, amelyet a unit neve követ. A Delphi az újonnan megnyitott unitoknak a Unit1, Unit2 stb. neveket adja. Amennyiben egy unitból más unitokat is el szeretnénk érni, akkor azt a unit **uses** sorában fel kell tüntetni.

Megjegyzés: A unit nevét kizárólag a File | Save as ponton keresztül változtathatjuk meg, máskülönben a projekt összezavarodhat.

A unit azt a nevet kapja, amely alatt a unit fájlt elmentjük. Természetesen a neveknek egyedieknek kell lenniük, a Delphi nem is engedi, hogy több egyidőben használt unitnak ugyanaz legyen a neve. Még az sem

megengedett, hogy egy unit neve megegyezzen a projekt nevével. A neveknek meg kell felelniük a Dos fájlnevekre vonatkozó standard nyolckarakteres konvenciónak a Delphi 16-ban, a 32 bites verzióban pedig a 255 karakteres korlátozásnak. A unit fájlok .PAS kiterjesztéssel mentendők el. Ha egy unitot pl. az ElsőUnitom név alatt mentünk el, akkor azt minden **uses** blokkban az ElsőUnitom néven kell feltüntetni.

Az interface blokk

Az interface blokk **uses** sora tartalmazza azokat a külső unit neveket, amelyeket a projektben használni kívánunk. A Delphi automatikusan feltünteti itt a standard unitok neveit, mint például a SysUtils és Controls. Ha új komponenseket adunk formhoz, a Delphi beszúrja a megfelelő neveket a form **uses** sorába. A hozzáadott unitok törlésébe a Delphi nem szól bele még akkor sem, ha az azokat használó komponenseket letöröltük a projektből.

Megjegyzés: *Az általunk létrehozott unitok interface blokkjához a Delphi nem adja hozzá a* ***uses*** *sort – erről nekünk kell gondoskodnunk.*

Miként arról már korábban volt szó, az **interface** blokk az a hely, ahol a globális constansokat, típusokat, változókat, függvényeket (azaz minden olyan dolgot, amit a unitot használó többi unit számára elérhetővé szeretnénk tenni) felsoroljuk. Az **interface** blokkban – ahogy azt korábban láhattuk – a globális függvényeknek és eljárásoknak csak a fejrészét tüntetjük fel.

Pascal tipp: *Amennyiben egy a Borland Pascal for Window korábbi verzióiban íródott unitot szeretnénk használni, amely a WinCrt unitot használja, akkor lényeges, hogy a* ***unit*** *uses sorában szerepeljen a WinCrt név. Annak ellenére, hogy néhány régebbi Delphi 16 dokumentáció azt állítja, a WinCrt-t a Delphi nem adja automatikusan a form unitjának* ***uses*** *sorába.*

Tisztában kell lennünk azzal is, hogy a Delphi a form vizuális részeinek szükséges deklarálásait hozzáadja a formhoz tartozó unit interface

blokkjához (l. még a 7. fejezetet). Nagyon ritkán kerülünk olyan helyzetbe, hogy meg kelljen változtatnunk az interface blokkban azokat a részeket, amelyeket a Delphi töltött fel információkkal.

Az implementation blokk

Az implementációs blokk az a hely, ahová a függvények és eljárások kódja kerül, függetlenül attól, hogy deklaráltuk-e őket az **interface** blokkban vagy sem. Az itt deklarált Delphi objektumok kizárólag a uniton belül lesznek elérhetők. Egy form unitjának az implementációs blokkját az alábbi fordítási direktíva vezeti be:

```
{$R *.DFM}
```

Ügyeljünk arra, nehogy ezt a sort véletlenül is letöröljük. Ez a direktíva jelzi a Delphinek, hogy a unithoz vizuális rész is társul, és a külső (pl. DLL) hivatkozások feloldása is itt történik.

Az Initialization blokk

Ez a blokk opcionális (használata nem kötelező) és nem is adódik magától a unithoz, amikor azt először megnyitjuk. Azok az utasítások kerülnek ebbe a részbe, amelyek végrehajtása a unit inicializációjához szükséges. A blokkot az **initialization** kulcsszó vezeti be. A Delphi abban a sorrendben inicializálja a unitokat, amilyen sorrendben azok a főform **uses** sorában vannak fölsorolva.

Pascal tipp: *A Delphi felismeri a régebbi* begin *kulcsszót is az új* ***initialization*** *mellett, de ez megzavarhatja az IDE-t, használatát ezért lehetőleg kerüljük.*

■ *Lényeges tudnivalók a fordítóról*

Mielőtt a prokteket lefordítjuk, hogy azokból önmagukban lefuttatható alkalmazásokat hozzunk létre, különböző beállításokat tehetünk meg, amivel a generált kód egyes jellemzőin változtathatunk. A leggyakrabban

használt fordítási direktívák megtalálhatók a Project Options dialógusablak Compiler oldalán (a Delphi 16-ban Options | Project). A különböző direktívák más-más szituációban lehetnek hasznosak. Például a *Pentium Safe FDIV* (Pentium biztonságos FDIV műveletek) a régebbi, hibás Pentium processzoros gépeken kikerüli a hibás aritmetikai utasításokat. A *bővített szintaxis* a PChar változók és a nulla-végű (nagy) sztringek használatakor jön jól. Sok direktívát – többek között a dialógusablak Debugging részben találhatókat – általában bekapcsolunk fejlesztés alatt, de a végső fordítás előtt kiiktatjuk őket, mert használatuk a sebesség rovására megy, és a program memóriaigénye is megnőhet.

Megjegyzés: Arra is lehetőségünk van, hogy közvetlenül a kódba helyezzünk fordítási direktívákat a megfelelő kapcsoló (l. beépített help) kapcsos zárójelek közé helyezésével. Az első zárójel után egy $ jel kell hogy álljon. A fordítási direktívákat általában a fordítandó unit deklarációs része elé kell helyezni. A + jel használatával bekapcsolhatjuk, míg a – jel használatával kikapcsolhatjuk az egyes direktívákat. Például az {$X+} bekapcsolja a bővített szitaxist, a {$X–} pedig megszünteti.

A Delphi lehetőséget kínál a projektek dinamikusan szerkeszthető könyvtárként (DLL-ként) való lefordítására, ezáltal biztosítjuk, hogy azok más Windows programok számára is elérhetőkké váljanak. Ehhez azonban a DPR projekt fájljának a megváltoztatására van szükség:

1. A **program** kulcsszót cseréljük ki a **library** kulcsszóra.
2. A DPR fájl uses sorából töröljük a hivatkozásokat a Forms unitra.
3. A begin/end pár között ne hagyjunk semmit.
4. A begin/end blokk elé szúrjuk be az **exports** kulcsszót.
5. Adjuk a uses sorba azoknak a unitoknak a nevét, amelyek függvényeit vagy eljárásait exportálni kívánjuk (elérhetővé kívánjuk őket tenni).
6. Soroljuk fel az összes exportálandó függvényfejet az **exports** kulcsszó után.

Az itt tisztázatlan részletekről a beépített súgó „Compiling a Project into a DLL" és a „Component Writers Guide" részei adnak felvilágosítást.

Gyakori fordítási direktívák

Ebben a részben a Compiler-oldalon elérhető gyakran használt fordítási direktívák közül azokat mutatjuk be, amelyek mindkét Delphi változatban megtalálhatók.

Default jelölő négyzet. A Default választása esetén a Delphi elmenti a fordítással kapcsolatos beállításokat, a későbbi fordításokat ezeknek megfelelően fogja végezni.

Pentium-Safe FDIV (Pentium biztonságos FDIV műveletek). Ez a direktíva a korábbi Pentium processzorok hibás FDIV utasításait megkerülő, lebegőpontos kód programba építését vezérli. A $U kapcsolónak felel meg.

Range Checking (értéktartomány ellenőrzése). Sztring- és tömbindexek esetén értékhatár-ellenőrzést végez. A direktívával lefordított program futása lassabbá válik, ezért leggyakrabban csak a fejlesztési ciklusban használják. A $S kapcsolónak felel meg.

Stack Checking (verem túlcsordulásának ellenőrzése). Az így lefordított programba beépített kód ellenőrzi, hogy elegendő hely áll-e rendelkezésre a veremben a lokális változók számára az egyes eljárások hívásakor vagy rekurzió használatkor. Ez szintén lassabb futáshoz vezet, ezért inkább csak fejlesztés közben használjuk. A $S kapcsolónak felel meg.

I/O Checking (input-output műveletek ellenőrzése). Az így lefordított programba beépített kód minden B/K hívás után ellenőrzi az esetleges hibákat. Jobb a kivételkezelés használata (l. 8. fejezet). A $I kapcsolónak felel meg.

Overflow Checking (túlcsordulás ellenőrzése). Az így lefordított programba egész-műveletek túlcsordulását ellenőrző kód épül be. Használata fejlesztési időben ajánlott. A $Q kapcsolónak felel meg.

Strict Var-Strings (Sztringtípusú változó paraméterek ellenőrzése). A bekapcsolt direktívával fordított program engedélyezi egy 20 karakterből álló sztring átadását egy olyan eljárásnak, amely egy 255 karakterhosszúságú sztringet vár. A $V kapcsolónak felel meg. Az Open parameters

(nyitott sztring paraméterek) direktíva bekapcsolásával hatása semmissé válik.

Complete Boolean Eval (logikai kifejezések kiértékelése). Általában ez az opció ki van kapcsolva. Bekapcsolt állapotban a Delphi a logikai kifejezéseket teljes egészében kiértékeli függetlenül attól, hogy a végső eredmény ismert-e már. Például amikor az and operátort használjuk két kifejezés összekapcsolására és az első kifejezés hamis, akkor nincs szükség a második kiértékelésére. A $B kapcsolónak felel meg.

Extended Syntax (bővített Delphi szintaxis). Ez szükséges a PChar támogatásához (lásd a fejezet „Nulla-végű sztringek és a PChar" részét). Bekapcsolt állapotban a függvényeket eljárásként hívhatjuk meg, a függvény visszatérési értékét nem vessszük figyelembe. A $X kapcsolónak felel meg.

Typed @ operátor (mutatók típusának ellenőrzése). Bekapcsolt állapotban a Delphi megvizsgálja, hogy a mutatókkal végzett értékadások értelmesek-e. Ellenőrzi a @ operátor által visszaadott típust. A $T kapcsolónak felel meg.

Open Parameters (nyitott sztringparaméterek). Bekapcsolva, a fordító engedélyezi a nyitott sztringparamétereket függvény- és eljáráshívások esetén. A $P kapcsolónak felel meg.

Debugging Information (nyomkövetési információk). Fejlesztés alatt kapcsoljuk be. Végső fordítás előtt feltétlenül kapcsoljuk ki. Részletesebben erről a direktíváról a 9. fejezetben lesz szó.

További fordítási direktívák a Delphi 32-ben

Itt röviden tárgyaljuk a legfontosabbakat azon direktívák közül, amelyek a Delphi 32-ben elérhetők (a Compiler-oldalon), de a Delphi 16-ban nem.

Optimalization (optimalizálás). Alapértelmezésben be van kapcsolva. Végső fordítás előtt mindenképp kapcsoljuk be. Bekapcsolva a Delphi 32 rendkívül jó optimalizációt végez a kódon. Kikapcsolásával csökken a fordítási idő.

Aligned Record Fields (rekordmezők igazítása tárolási határokra). Ez a direktíva az adatstruktúrák elemeinek tárolását a 32 bites határokhoz igazítja. A $A kapcsolónak felel meg.

Assignable Typed Constants (értékadás típusos konstansnak). Használjuk a Delphi 16 kódokkal való kompatibilitás biztosítására. Bekapcsolt állapotban a fordító engedélyezi az értékadást a típusos konstansoknak. A $S kapcsolónak felel meg.

Huge Strings (Nagy sztringek). Bekapcsolásával a Delphi a sztringeket nagy sztringekként kezeli.

Show Hints (útmutatás). Bekapcsolt állapot esetén útmutatásokat kapunk.

Show Warnings (figyelmeztetés). Bekapcsolásával lehetővé tesszük a figyelmeztető üzenetek megjelenését.

További direktívák a Delphi 16-ban

Az alábbiakban közöljük a Delphi 16 Compiler oldalán található fontosabb fordítási direktívákat, amelyek a Delphi 32-ben nem érhetők el.

Force Far Calls (FAR módú rutinhívások). Meglehetősen ezoterikus kapcsoló, amely létét kizárólag az Intel processzorok szegmentált architektúrájának köszönheti. Lényegében nincs olyan körülmény, amely ennek a bekapcsolását igényelné. A Delphi fordító elég intelligens ahhoz, hogy tudja, mikor kell far módú hívásokat használnia. A $F kapcsolónak felel meg.

Word Align Data (Adatok igazítása tárolási egység határára). Ez, a kód megnövelésével biztosítja az adatok gyorsabb kezelését. Biztosítja, hogy az adatok páros memóriacímre igazítva kerüljenek tárolásra. Az Intel processzor architektúráját jellemző bizonyos furcsaságoknak köszönhetően, a direktíva bekapcsolása gyorsíthat a program futásán. Nem árt kipróbálni mindkét állást a végső fordítás előtt. A $A kapcsolónak felel meg.

Smart Callbacks (gyors visszahívás). Ez mindig legyen bekapcsolva! Ellenkező esetben Windows üzenetek sorával találjuk szembe magunkat. A $K kapcsolónak felel meg.

Windows Stack Frame (Windows veremkeret). Ezt hagyjuk kikapcsolva, hacsak olyan kódot nem kívánunk létrehozni, amely Windows 3.0 alatt is fut esetleg (azért „esetleg", mert véleményünk szerint nem sok időt töltöttek a Delphi ilyen irányú fejlesztésével). $W-nek felel meg.

Pascal megjegyzés: A kapcsolóra szükség lehet a régebbi OWL alapú programok fordításánál.

Feltételes fordítás

Lehetőségünk van egy alkalmazásról két verziót létrehozni anélkül, hogy számukra két külön forráskódot kellene fenntartani. Ehhez az úgynevezett feltételes direktívák nyújtanak segítséget. Így megoldhatóvá válik például egy nemzetközileg használt program különböző nyelvű változatainak összefogása egy forráskódban, a neheze – azaz a különböző verziók előállítása – a Delphi fordítójára maradna. A folyamatot magát *feltételes fordításnak* nevezzük. A feltételes fordítás vezérlése a programozó által definiálható feltételes szimbólumok alkalmazásával történik. Alábbiakban egy egyszerű példát láthatunk:

```
{$DEFINE FRENCH}
{$IFDEF FRENCH}
ShowMessage ('Bon-jour.');
{$ELSE}
ShowMessage ('Jó napot kívánok!');
{$ENDIF}
```

Itt a feltételes szimbólumot a FRENCH jelenti, amelyet az első sor definiál. Így a képernyőn a „Bon-jour" köszöntés jelenik meg. A {$DEFINE FRENCH} nélkül a magyar nyelvű változat jelent volna meg.

Hasonlóan hagyhatunk hibakereséshez használt programrészletet is a forráskódban, amelynek lefordítását a HIBAKERESÉS feltételes szimbó-

lumhoz kötjük. Ekkor, ha ezt az extra részt is le akarjuk fordítani, akkor az alábbi sort iktatjuk a forráskódba:

```
{$DEFINE HIBAKERESÉS}
```

Az alábbi táblázat foglalja össze a feltételes direktívákat:

Feltételes direktíva	*Hatása*
$DEFINE	Definiálja a feltételes szimbólumot, és igazra állítja.
$ELSE	Az utána lévő kódrészlet akkor fordítódik le, ha a feltételes szimbólum hamis.
$ENDIF	Lezárja a feltételes fordítást.
$UNDEF	Egy előzetesen igazra állított feltételes szimbólumot hamisra állít.

Megjegyzés: *Sajnálatos módon a Delphi 32 különböző pozitívumainak kiaknázása rendkívül nehéz feltételes fordításon keresztül. Általában az olyan objektumok használata, mint például a nagy sztringek, túl sok feltételesen fordítandó kódrészletet kívánnak meg, ami már nem lenne praktikus.*

■ *A goto utasítás*

Mint a legtöbb programozási nyelv, a Delphi is rendelkezik a feltétel nélküli ugrást megvalósító utasítással: a goto-val. A modern programozók három csoportra oszthatók: azokra, akik nem tudják és nem is törődnek azzal, mikor lehet goto-t használni, azokra akik ugyan nem tudják, de nagyon óvatosak, és végül azokra akik tudják.

Az világos, hogy a goto meggondolatlan használata spagettiszerű programhoz vezet: nehezen olvasható, nehezen javítható kódhoz. Vannak azonban esetek, amikor a goto használata a kódot érezhetően tisztábbá és könnyebben érthetővé teszi. (A Delphiben ilyen esetekkel akkor találkozhatunk, amikor többszörösen egymásba ágyazott ciklusok belsejében vagyunk, és valamilyen feltétel teljesülése a ciklusok elhagyására kényszerít minket. Nem használhatjuk a **break**-et, hiszen azzal az aktuális ciklusból csak egy „héjjal" kerülnénk kijjebb.

A goto használatához a Dephiben először szükséges egy címkét definiálnunk a **label** kulcsszóval, amelyet a címke azonosítója követ – ha több címkét definiálunk egyszerre, akkor azokat vesszővel választjuk el egymástól. Mint minden deklarációt, a címkét deklaráló sort is pontosvessző kell hogy zárja. Például:

```
label
  RosszInput, RosszOutput;
```

A címkéket a programtörzs előtt definiáljuk, és természetesen rájuk is érvényesek a Delphi azonosítókra vonatkozó szabályok. Használhatunk ugyan számokat is 1 és 9999 között, de jobban tesszük, ha jelentéssel bíró cimkéket használunk.

A deklarált címkéket a deklarációt követő programtörzs bármelyik részében felhasználhatjuk (a unitból nem ugorhatunk ki). Címkét így használhatuk:

```
RosszInput:
  {A végrehajtásra kerülő kód helye};
```

Tegyük fel például, hogy adatok bekérésénél egymásba ágyazott **for** ciklusokat használunk, a beadott adatsor végét az jelzi, ha a felhasználó három Z betűt ír be, ekkor a vezérlés átadódik az adatokat feldolgozó résznek:

```
var
  i, j: Integer;
  AdatokBe: String;
  cellák : array [1..10, 1..100] of string[20];
label
  RosszInput;
begin
  for i := 1 to 10 do
    for j:= 1 to 100 do
      begin
        AdatokBe := InputBox('Adatbekérés', 'Írja
        be az adatokat, a ZZZ az utolsó adat', '')
```

```
        if GetData = 'ZZZ' then
          goto RosszInput
        else
          Cellák [i, j] := AdatokBe;
      end;
RosszInput:
  ShowMessage (´Adatbevitel megszakítva felhasználói
  kérésre´);
```

A **break** kulcsszó használatával ugyanezt megvalósítani nehézkes lenne: sok plusszmunkát igényelne a csatolt ciklusból a teljes kilépés.

7. FEJEZET

Objektumok

Mostanra valószínüleg eljutottunk addig a pontig, hogy ismerjük a Delphi beépített objektumainak manipulálásához szükséges alaptechnikákat. Ahhoz, hogy a Delphi képességeit jobban ki tudjuk használni, fontos megismerkednünk az objektumorientált programozási modell Object Pascalbeli megvalósításával. Az objektumorientált programozás egyre nagyobb tért hódít, és úgy tűnik, hogy mára domináns programozási módszerré nőtte ki magát, felváltva a strukturált programozási technikákat, amelyeket a 70-es évek elején dolgoztak ki. Ha az olvasó még nem dolgozott objektumokkal, akkor valószínüleg most széttárja a karját, és azt kérdezi, miért ez a nagy felhajtás. A fejezetnek a célja éppen ennek megmutatása (legalább bevezető szinten). Az első oldalakon azokkal a definíciókkal és a fogalmakkal ismerkedhetünk meg, amelyek nélkülözhetetlenek az OOP (objektumorientált programozás) tárgyalásához. A terminológiai bevezető után tárgyaljuk majd az OOP-t a Delphiben.

Az objektumorientált programozásról vég nélkül lehetne írni. Egy kiadott könyveket felsoroló katalógusból kitűnik, hogy több mint 150 könyv látott napvilágot az objektumorientált programozásról, és ez még csak a kezdet: hetente jelennek meg a témával foglalkozó könyvek. A Delphiben megvalósított OOP teljes leírása méreteit tekintve legalább még egy ugyanilyen könyvet ölelne fel. Ez a fejezet valójában csak arra elegendő, hogy ízelítőt adjon az objektumorientált programozásból. Egyebek között kimaradt a saját vizuális komponensek készítési módjainak ismertetése. Ehhez a Delphihez mellékelt *Component Writer´s Guide* nyújt segítséget (amely kb. ennek a könyvnek a felét teszi ki, és sajnos még így is meglehetősen vázlatos).

■ *Bevezetés*

Kezdjük egy kérdéssel, aminek első látásra semmi köze nincs a programozáshoz: Hogy válhatott a Gateway 2000 egy több milliárd dolláros tőkével rendelkező nagyvállalattá gyorsabban, mint bármely más cég az amerikai történelemben? A legtöbb ember azt mondaná, hogy jó számítógépeket készítettek, és azokat kellően alacsony áron tudták árulni. És hogyan csinálták ezt? Nos, a kérdésre a válasz egyszerűsített formája valahogy így néz ki: megbízható forrásból komponenseket vásároltak, majd összeszerelték azokat saját elképzelésüknek megfelelően. Nem fektettek pénzt tápegységek, lemezegységek, alaplapok stb. tervezésébe és előállításába. Ennek köszönhetően olyan minőségi termékeket tudtak gyártani, amelyek alacsony árai állták a versenyt.

Nem csináltak tehát mást, mint különböző funkciókat megvalósító, előregyártott részegységeket vásároltak. Amikor például egy tápegységet vettek, valami olyanra tettek szert, aminek bizonyos tulajdonságai (méret, alak, kimenő feszültség nagyság stb.) és funkciói (stabilizált feszültség kimenet, szünetmentesség stb.) voltak. Az objektumorientált programozás is ezen az elven működik. A programot különböző objektumok építik fel, amelyek mindegyikének megvannak a saját tulajdonságai és funkciói. Természetesen a dolog egy kis bizalmat is igényel: el kell hinnünk, hogy az objektumok egymással és programunk többi részével biztonságosan kommunikálnak, hiszen mindez számunkra nem látható. Az, hogy az objektumokat megvesszük-e vagy inkább magunk készítjuk el őket, részben a pénztárcánk méretétől, részben pedig attól függ, hogy mennyi szabadidőnk van. Mindkét esetben, ameddig az objektumok megfelelnek az elvárásainknak, lényegtelen az, hogy belül milyen módon valósítják meg a feladatokat. Az objektumorientált programozás lényege éppen az, hogy csak azzal kell törődnünk, ami látszik. Hasonlóan ahhoz, ahogy a Gateway nem törődött a tápegység belsejével, amíg az azt csinálta, amit ők akartak, a legtöbb programozónak sem szükséges azon rágódnia, hogyan működik a TButton komponens, amíg az ellátja a feladatát. Miként azt tapasztalhattuk, a Delphi objektumaira szinte teljes mértékben rábízhatjuk magunkat!

A produktivitás titka az objektumorientált programozásban az, hogy törekedni kell objektumaink lehető legteljesebbé tételére, valamint arra, hogy minél több más objektum és programrész felé mutassanak csatlako-

zási pontokat. Az OOP zsargonjában ezt úgy mondják, hogy lehetővé kell tenni a kliensek és az objektumok közti komunikációt. Ha az objektumokat úgy tervezzük, hogy a feladatkörükbe tartozó összes üzenet vételére és az azok által szállított adatok feldolgozására is képesek legyenek, akkor használhatóságuk megn és a hibajavításhoz szükséges idő csökken.

Láthattuk, hogy a Delphi beépített objektumai megfelelnek ezeknek a kritériumoknak. Ezek az objektumok funkciók széles skáláját látják el, természetesen így is adódhatnak olyan programozási feladatok, amelyek más objektumok használatát is megkövetelik. Ekkor két lehetőség kínálkozik: az egyik, hogy megvesszük egy erre specializálódott cégtől, a másik, hogy magunk készítjük el. Mielőtt döntenénk, mindig célszerű figyelembe venni az elkészítéshez szükséges idő és a beszerzési ár arányát. Gyakran nem éri meg feltalálni a spanyolviaszt. A MicroHelp-hez hasonlóan, sok kis cég kínál minőségi termékeket, viszonylag jutányos áron. Én is rendszeresen használok külső komponenseket programjainkban, és általában elégedett vagyok az eredménnyel (főleg ha a forráskódot is csatolták, amelynek segítségével, szükség esetén kisebb változtatásokat is végrehajthatunk a komponensen).

Ha mégis arra kényszerülnénk, hogy saját objektumokat írjunk, akkor az OOP egy másik fontos jellemzője ezt nagy mértékben megkönnyíti számunkra: az objektumokat egymásra lehet építeni. Amikor így egy új objektumot hozunk létre, akkor az örökli a szülőjének összes függvényét és tulajdonságát, amelyeket kívánság szerint módosíthatunk, vagy akár törölhetünk is. Valószínű, hogy a Delphi sokszínű objektumkínálatára és az esetleges külső eszközökre építve nem kerülünk olyan helyzetbe, hogy vizuális objektumokat kellene létrehoznunk. Az már inkább elképzelhető, hogy magunk hozunk létre bizonyos nem vizuális komponenseket. Például egy NagySztring objektum megalkotása, amelynek függvényeivel a Delphi 16 nagy sztring kezelését végezzük, elég kézenfekvőnek tűnik. Az ilyen és ehhez hasonló objektumoknak az elkészítése megéri a ráfordított időt és fáradságot.

Az OOP terminológiája

A hagyományos strukturált programozás alapja az adatstruktúrák megtervezése, és az azokkal történő különböző műveletek elvégzése a megfelelő függvényeken keresztül. Ezt az utóbbi, általában végetérő folyamatot hívják algoritmusnak. Ezért adta Nikolaus Wirth, a Pascal nyelv atyja, programozásról szóló könyvének az *Algorithms + Data Structures = Programs* (Prentice Hall, 1976). címet. Vegyük észre, hogy a címben az algorithms (algoritmusok) szó szerepel elsőként és a data structures (adatszerkezetek) kifejezés másodikként. Ez hűen tükrözi az akkori programozói felfogást. Először azt döntötték el, hogy milyen módon kezelik az adatokat, és csak azután került sor a megfelelő adatszerkezet megalkotására, amellyel a tervezett műveletek könnyebben voltak elvégezhetők. Az OOP-ban az adatszerkezet és az algoritmus teljesen egyenrangú.

A fejezetrész hátralevő részében az OOP alapterminológiáját ismertetjük. Ez nem kis mennyiség, de két okból is érdemes megtanulni. Az egyik, hogy a fejezet további részének megértése nem képzelhető el enélkül, a másik, hogy az OOP zsargon ismerete nélkül nem leszünk képesek a Component Writer´s Guide-ot elolvasni, ami pedig fontos a további OOP ismeretek megszerzéséhez.

Az osztályok (Classes)

Az osztályok általában olyan sablonoknak tekinthetők, amelyekből a tulajdonképpeni objektumok előállnak. Gyakran szokták az osztályokat süteményformákhoz hasonlítani, ahol az objektumok a sütemények. A lefoglalandó memóriát a sütemény „tésztájának" képzelhetjük. A Delphi arra törekszik, hogy a „tésztakészítés"-i fázist elrejtse előlünk. Amikor egy osztályból létrehozunk egy objektumot, akkor azt mondjuk, hogy elkészítettük az osztály egy *példányát*.

Zártság (Encapsulation)

Ez az objektumok egyik legfőbb tulajdonsága. Formálisan szólva a zártság nem más, mint az adatok és a viselkedés egységbe forrasztása. Néhányan szeretnek úgy gondolni az egyes objektumokra, mint egy speciális típusra, amely nem csak adatokat, hanem függvényeket és eljárásokat is tartalmaz.

Ez az új típus megfelel az objektum osztályának. Valójában ezek tükröződnek az objektumok létrehozásánál használt Object Pascal szintaxisban is (l. az „Objektumok létrehozása a Delphiben" fejezetrészt). (A Delphi objektumai tényleg nem mások, mint egy speciális típus képviselői!)

Megjegyzés: *Az objektum adatrészét az ún. adatmezők (fields) alkotják. Függvényeit és eljárásait együtt az objektum metódusainak (methods) hívják.*

A zártság működését azzal biztosítjuk, hogy a programokat soha (szinte soha) nem engedjük közvetlenül hozzáférni az objektum adatmezőihez. A program csak az objektum metódusain keresztül áll kölcsönhatásban az adatokkal. (A tulajdonságok adnak lehetőséget az adatokkal való direkt kapcsolatra, a zártság elvének megsértése nélkül.) A zártság kölcsönzi az objektumoknak azt a „fekete doboz"-szerű viselkedést, amely kulcsfontosságú az újra hasznosíthatóságukhoz és a hibakeresés hatékonyságához.

Öröklődés (inheritance)

Öröklődés az osztályok azon képessége, amely lehetővé teszi azok más osztályokból való származtatását. Az öröklődés lényege az, hogy segítségével specializált feladatokat megoldó kódot egyszerűbben tudunk előállítani. A származtatott osztály (vagy ritkábban alosztály) változói és metódusai öröklődnek az őstől. Ez a nyelv készítőinek is hasznára vált, hiszen így mindig volt egy alap, amire az új osztályokat építhették. Például a Delphi alkotói kezdetben egy általános szerkesztési ablak komponenst hoztak létre, amelyből kialakították a maszkolt szerkesztési ablakokat és a közönséges szerkesztési ablakokat. Még általánosabban igaz, hogy a Delphi objektumai egyetlen közös ős, a TObject leszármazottai. Azokat az osztályokat, amelyekre más objektumokat építünk, *ősosztályoknak* (ancestor vagy parent class) vagy *alaposztályoknak* (base class) nevezzük.

A Delphit használó programozó is nagy hasznát veszi az öröklődésnek. Többek között a Delphi tanulása is nehezebb lenne, ha a különböző tulajdonságok – pl. a Left és a Top – másként viselkednének különböző komponensek esetén. Minden látható komponens ugyanattól az őstől származik, ezért mindannyian öröklik a Left és Top tulajdonságokat. A

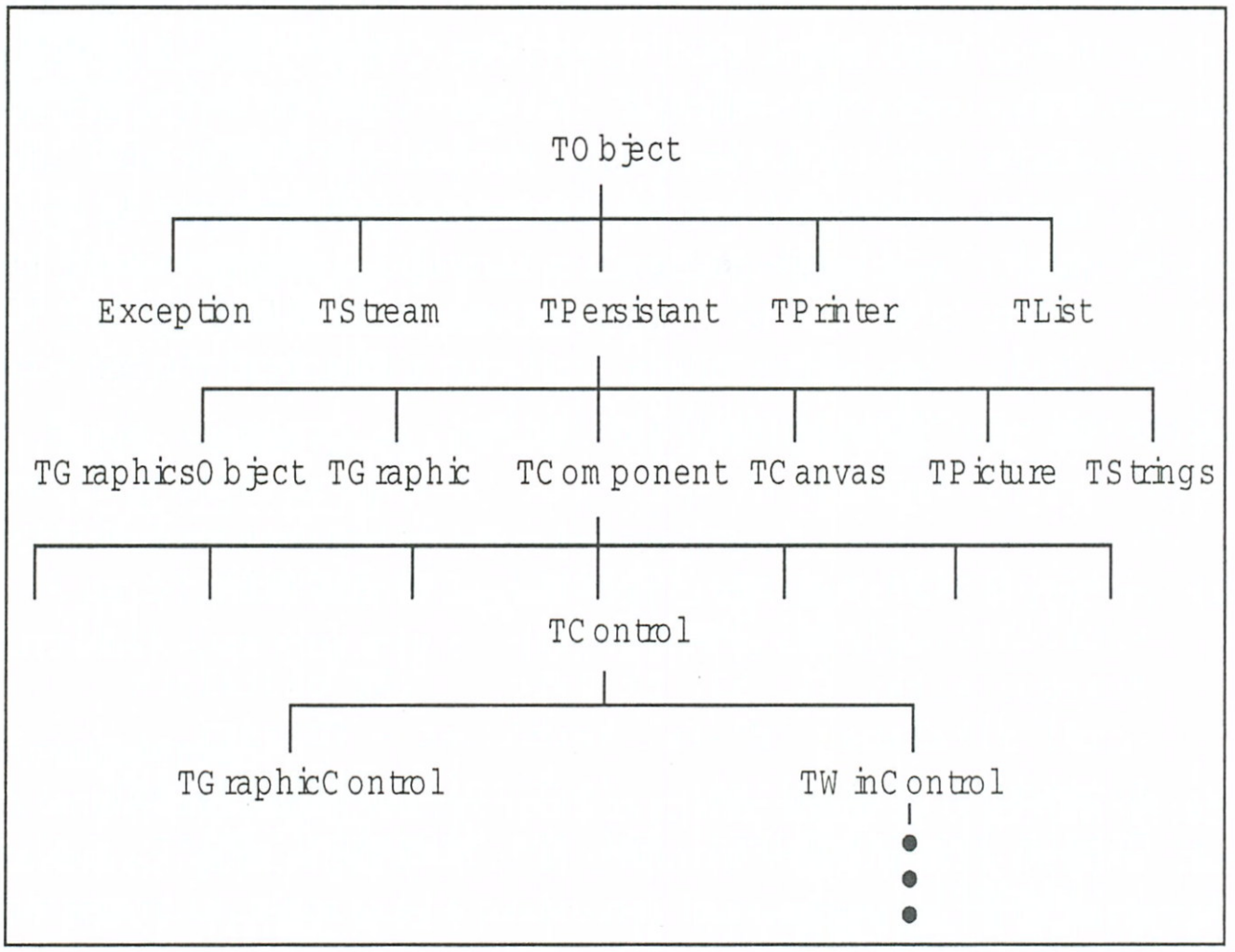
Delhi alkotói nem definiálták újra ezeket a tulajdonságokat az örökölt objektumokban, ezért azok teljesen egyformán viselkednek bármely komponensről legyen is szó.

Hatékonyságának egy másik oldala akkor válik nyilvánvalóvá, amikor saját komponensek építésébe kezdünk. Ha egy elképzelésünknek megfelelő szerkesztési ablakot kívánunk elkészíteni, akkor felhasználhatjuk az összes szerkesztési ablak ősét (TCustomEdit), amely tulajdonképpen egy lecsupaszított szerkesztési ablak, vagy annak egy gyermekét. Ha a megfelelő ős- vagy alaposztályt vesszük kiinduló pontnak, akkor a munka nagy részével tulajdonképpen kész is vagyunk. Az osztályok egymáshoz való viszonyát az *osztály-hierarchiája* írja le. Az osztályok hierarchiáját biológiai példával élve, az élőlények evolúciós rendszerezéséhez hasonlíthatjuk: az előgerinchúrosok törzséből származnak a gerincesek, amelyet a halak, a kétéltűek, a hüllők, a madarak és az emlősök osztálya alkot. A Delphi objektumainak hierarchiáját a 7-1. ábrán láthatjuk. Vannak helyzetek, amikor ennek ismerete hasznos lehet. A Delphi osztályhierarchiájáról még lesz szó az ObjectBrowser-ről szóló részben.

Általában az alosztályok használják (öröklik) a szülők metódusait is. Az OOP (és a Delphi) lehetővé teszi azt, hogy egy objektumban olyan név alatt definiáljunk új metódust, amilyen szerepelt az ősében. Ezt hívják *felüldefiniálásnak* (overriding). Egy valódi objektumorientált nyelv, mint a Delphi, a felüldeklarálásnál is hatékonyabb eszközt biztosít számunkra: a *többalakúságot* (polimorfizmust). A polimorfizmus lényege, hogy azonos hivatkozás esetén más-más metódus kerül végrehajtásra, attól függően, hogy a program futása közben melyik szintjén vagyunk az osztályhierarchiának. A többalakúságot bármilyen metódusra alkalmazhatjuk, amely egy alaposztályból öröklődött.

A polimorfizmus egyszerűsíti a programozó munkáját. Egy egy új objektum definiálásakor, amelyet egy már létező objektumra építünk, nem szükséges átalakítani az ős kódját, hogy figyelembe vegyük az új objektum jelenlétét. Tegyük fel például, hogy egy új objektumban egy örökölt metódust használunk (m1), amely meghív egy másik metódust (m2). Legyen továbbá az m2 definiálva mind az ős, mind a származtatott objektumban. Ekkor a polimorfizmus szerint (a megfelelő szintaxis használatával) az örökölt metódus az újabb m2 metódusnak adja át a vezérlést, de ha az ősön keresztül hívjuk meg a m1 metódust, a régebbi, m2 kerül végrehajtásra.

A többalakúságot az úgynevezet *késői kapcsolatteremtés* (late bindig) teszi lehetővé. Ez azt jelenti, hogy a fordító nem hozza létre a konkrét hívást megvalósító kódot a fordításkor, hanem egy olyan kódot ültet a programba, amely a hívó objektumhoz tartozó információkat használva határozza meg, hogy melyik metódusnak adódjon át a vezérlés.



7-1. ábra. A Delphi Visual Component Library-jának (VCL-jének) egyszerűsített objektum-hierarchiája

A késői kapcsolatteremtés az ún. virtuális metódusokon keresztül valósulhat meg.

Megjegyzés: *Minden virtuális metódushoz létezik az adatszegmensben egy úgynevezett virtuális metódus táblázat (VMT), a program futás közben ebből a táblázatból veszi a metódus címét. A dinamikus metódusokhoz az ún. dinamikus metódus táblázat (DMT) tartozik.*

■ *Objektumok létrehozása a Delphiben*

Tegyük fel, hogy a számítógépes kártyajátékok fejlesztői számára szeretnénk egy eszköztárat biztosítani. Nyilvánvalóan egy objektumra lesz szükségünk, amellyel egy pakli kártyát reprezentálhatunk. Az egyszerűség kedvéért ezt egy nem vizuális objektummal valósítjuk meg. Első megközelítésben, (az objektumok egyik nagy előnye, hogy használatukkal a megoldást egyszerűen finomíthatjuk, ami igen közel áll a programozói gondolkodásmódhoz) a kártyalapokat tároljuk egy 52 elemből álló sztringtömbben. Szükségünk lesz két eljárásra és egy függvényre. Az egyik eljárás inicializálja, a másik megkeveri a kártyákat, a függvény pedig kioszt egy kártyalapot. Ezt az objektumot megvalósító kód így néz ki: (Ennek a kódnak van néhány hibája, amelyeket a következő részben vizsgálunk meg – ez csak az alapstruktúrát írja le.)

```
type
  TDeckOfCards = class(TObject)   {A típusok
  általában 'T'-vel kezdődnek}
  Fcards: array [1..52] of Byte;   {Az osztályok
  mezői általában 'F'-fel kezdődnek - FCards}
  procedure Initialize;   {Ezt tulajdonképpen egy
  Create konstruktorral kellene megvalósítani -
  l. később}
  procedure Shuffle;
  function Deal(CardNumber: Integer): String;
end;
```

Megjegyzés: Amikor új objektumokat definiálunk a Delphiben, a definíció általában az ***Interface*** *blokk form deklarációt tartalmazó* ***type*** *része alá kerül. Az osztályokat a unit implementációs blokkjában is definiálhatjuk, ám ekkor azok csak a unitban lesznek használhatók.*

Vizsgáljuk meg közelebről ennek az objektumnak a deklarálását. Miként azt korábban említettük, az objektumdeklarációk szintaktikailag megfelelnek bármely más típusdeklarációnak, és általában a formdeklarációk után helyezkednek el. Az első sor utal arra, hogy az objektum

a Delphi melyik osztályától származik (itt ez a Delphi osztályok közös őse: a TObject):

ObjektumNév = class(TObject)

Pascal tipp: *Kompatibilitási okokból, az osztályok létrehozására a Delphi támogatja az* **Object** *kulcsszót. Jobban járunk azonban, ha a fenti szintaxist használjuk, amellyel biztosítjuk, hogy az objektumaink a TObject-től származnak. Ennek a származtatásnak egyik előnye, hogy a szükséges memóriát a Delphi a globális heap-ben és nem a lokálisban foglalja le az osztály objektumai számára, valamint így mutatóműveleteket spórolunk meg.*

Ezután következik az objektum adatainak (mezők), majd metódusainak (függvények és eljárások) deklarációja. Az objektum tényleges implementálásához ezen metódusokat kell megírnunk.

A metódusokat jelentő függvényeket és eljárásokat a unit implementációs blokkjában írjuk meg, ezzel definiáljuk az objektumot.

Konstruktorok

A fenti objektumban van egy speciális eljárás, amely az objektum inicializálását szolgálja. A példában ez az objektum létrejötte után történne. Habár semmi baj nincs ezzel a módszerrel, biztonságosabb, ha az adatmezőket (példánymezőket) az objektum létrejöttével egyidőben inicializáljuk. Ezáltal kikerüljük azt az eshetőséget, hogy az objektum későbbi használója létrehozza az objektumot, de elfelejti inicializálni. (Mivel az objektumok a Delphiben valójában mutatók, az eredmény egy rosszul inicializált mutató lenne – és valószínüleg egy „General Protection Fault" a Delphi 16-ban illetve egy „access violation" a Delphi 32-ben.)

A memóriaterület-foglalást az objektumpéldány számára és az adatmezők inicializálását egy speciális eljárás a *konstruktor* végzi, amelyet az objektum belsejében deklarálunk. A konstruktorok speciálisak az objektummodellben, mert ezek automatikusan meghívódnak az objektum létrejöttekor.

Mint minden más metódust, a kostruktort is az objektumban deklaráljuk, és csak az implementációs részben írjuk meg a kódját. A

konstruktor megírásának egyik módját az alábbi példa szolgáltatja, amelyik a TDeckOfCards objektumtípus módosított változata. A **constructor** kulcsszó jelöli a konstruktor metódust.

```
type
  TDeckOfCards = class(TObject)
  FCards: array [1..52] of Byte;   {Megjegyzés: az
  osztályok nevei általában 'F'-fel kezdődnek}
  Constructor Create; Virtual;   {virtuális
  metódusként deklaráljuk}
  procedure Shuffle;
  function Deal(CardNumber: Integer): String;
end;
```

A konstruktort megvalósító kódot is a **constructor** kulcsszóval vezetjük be.

```
constructor TDeckOfCards.Create;
var
  I: Integer;
begin
  for I :=1 To 52 do
    FCards[I] := I;
end;
```

Megjegyzés: A konstruktoroknak lehetnek paraméterei.

Az objektumokkal kapcsolatos nehézségek egyikét annak eldöntése okozza, hogy milyen paramétereket használjunk metódusaiban. Például lehet hogy sokan az olvasók közül úgy gondolják, hogy a konstruktornak tartalmaznia kellene egy FCards típusú paramétert. Ez nem lenne helyes, sőt mi több, hibaüzenetet kapnánk a fordítótól. A magyarázat abban keresendő, hogy objektumok metódusainak (beleértve a konstruktort is) hívásakor az objektum mezői automatikusan átadódnak (változó) paraméterként. Ezért a metódusban minden mezőhöz közvetlenül hozzáférhetünk.

Megjegyzés: Az objektumok megszüntetéséről a destruktorok gondoskodnak. Ezek szintén eljárások, meghívásukkal az objektumok számára lefoglalt memóriát szabadítjuk fel.

A kártyapakli objektum további finomítása

Bizonyos megfontolások után beláthatjuk, hogy célszerűbb az egyes kártyalapokat külön objektumokként kezelni. Ily módon később egyszerűen tudunk változtatni definíciójukon (később esetleg vizuálissá is tehetjük őket). Az új Deck objektumban a kártyalapok példányaiból alkotott tömb szerepelne. (Objektumok adatmezői lehetnek maguk is objektumok, de a szülő objektumnak feladata a mezőként szereplő objektumok dinamikus létrehozása és felszabadítása.) Az objektumunkban az első eljárás helyét a konstruktor foglalja el, továbbá szükségünk van egy másik eljárásra, amely a kártyák megkeverését végzi, és egy függvényre, amely egy lapot kioszt.

Az objektum típusdeklarációit ekkor az alábbi kóddal végezhetjük:

```
type
  TCard = class(TObject);
  public   {az így deklarált metódusok az uniton
  kívülről is elérhetőek}
  FValue: Byte;
  FSuit: String[8];
  constructor Create(V: String; S: String); virtual;
 end;
type
  TDeckOfCards = class(TObject)
  private
    FCards : array [0..51] of TCard;
  public
    constructor Create; virtual;
    destructor Destroy; override;
{   procedure Shuffle; még nincs kész}
    procedure Deal(Const I: Integer);
 end;
```

Vegyük észre, hogy az objektum ezen javított változata tartalmazza az opcionális **public** kulcsszót. A public kulcsszó alatt deklarált dolgok más unitokból is elérhetők lesznek. (A **private** szekció alatt deklarált eljárások nem lesznek láthatók más unitok részére.) Ez azt jelenti, hogy amíg egy objektumot az őt tartalmazó unitban kívánunk használni, addig nincs különbség a **public** és a **private** kulcsszó hatása között. A nyilvános módon (**public** kulcsszóval) deklarált adatokra és metódusokra történő hivatkozáskor a szintaxisban szerepel az objektum vagy form neve. Egy másik unitból való híváskor szükséges, hogy feltüntessük a metódust tartalmazó unit nevét is, amelyet egy ponttal választunk el a hivatkozás többi részétől. Általánosan:

UnitNév.ObjektumNév.NyilvánosMetódusNév

Megjegyzés: *Mindkét Delphi verzióban létezik két másik, a metódusok deklarálásánál megadható kulcsszó: a* ***protected*** *(védett) és a* ***published*** *(publikált). A védett metódusok csak az adott osztály és az abból származtatott osztályok által használhatók. A* ***published*** *kulcsszót lényegében csak komponensek készítésekor használjuk – így erről további részleteket a „Components Writer's Guide" megfelelő részéből tudhatunk meg. (A Delphi 32-ben ezenkívül létezik egy* ***Automated*** *metódusfajta, hatását megtudhatjuk a beépített help-ből.)*

Definiáláskor a metódusok nevében, az objektumazonosítót és a metódusazonosítót egy ponttal választjuk el. Az objektumtípus és a metódusok definiálása nem elegendő. Az objektum használatához először létre kell hozni azt, amit a Delphiben a konstruktor meghívásával érhetünk el. Az alábbi kód a pakli osztály egy példányát hozza létre, a helyes inicializálást egy nyomógomb megjelenítésével jelzi.

```
type
  TCard = class(TObject)
  public
   FValue: String;
   FSuit: String[8];
   constructor Create(V : String; S: String);
  end;
```

```
type
  TDeckOfCards = class(TObject)
   public
      Cards : array [0..51] of TCard;
      constructor Create; Override;
      destructor Destroy; Override;
{     procedure Shuffle; (nincs még kész}
      procedure Deal(const I: Integer);
   end;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
  public
      MyDeckOfCards: TDeckOfCards;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

constructor TCard.Create(V: String; S: String);
begin
  inherited Create;
  FValue := V;
  FSuit:= S;
end;

constructor TDeckOfCards.Create;
var
  I: Integer;
  Value: String;
```

```
  Suit: String;
  SuitInteger: Integer;
  C: TCard;
begin
  inherited Create;
  for I := 0 to 51 do
     begin
       Value := IntToStr(I mod 13);
       SuitInteger := I div 13;
       case SuitInteger of
         0: Suit := 'Treff';
         1: Suit := 'Káró';
         2: Suit := 'Kör';
         3: Suit := 'Pikk';
       end;
       C := TCard.Create(Value, Suit);
       FCards[I] := C;
     end;
 end;

destructor TDeckOFCards.Destroy;
var
  I: Integer;
begin
  for I := 0 to 51 do
    FCards[I].Free;
end;

procedure TDeckOfCards.Deal(const I: Integer);
var
  TestString: String;
begin
   TestString := FCards[I].FSuit + ' of ' +
   FCards[I].FValue;
   ShowMessage(TestString);
end;
{gyors ellenőrző eljárás}
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;

begin
  I := 5;
  MyDeckOfCards := TDeckOfCards.Create;
  MyDeckOfCards.Deal(I);
  {Az ehhez hasonló objektumokat általában a form
  OnCreate eseménye hozza létre és az OnDestroy
  eseménye szünteti meg.}
end;

end.
```

Vegyük észre, hogy először mindig a változókat deklaráljuk. Azáltal, hogy azokat a TForm1 **public** szekcióban deklaráljuk, biztosak lehetünk, hogy addig „élnek", amíg a form nyitva van.

```
public
  MyDeckOfCards: TDeckOfCards
```

A szintaxis, amellyel egy ilyen objektumpéldányt tudunk gyártani, a következő:

```
MyDeckOfCards := TDeckOfCards.Create;
```

A Create metódus utasítja a Delphit, hogy az objektumnak memóriát foglaljon. (A Delphi objektumai valójában mutatók.) Ezért fontos, hogy felszabadítsuk az elfoglalt memóriaterületet, amikor az objektumra már nincs szükségünk. Ezt a Free metódussal tehetjük meg (a kódban ezt a destruktor metódusokban találjuk).

__Megjegyzés:__ Amikor élesben programozunk, fontos hogy a Free metódus mindenképp végrehajtódjon. Ehhez a Create/Free metódusokat egy __try/finally__ blokk belsejében kell elhelyezni, a 8. fejezetben ismertetett módon.

■ *A Visual Component Library objektumai*

Bizonyára nagy meglepetést okozott mindenki számára, hogy a Delphi megváltoztatta a kódot, miután új komponenseket illesztettünk a formra. Legtöbbször akkor tapasztalhatunk ilyen változásokat, amikor a Visual Components Library-ból (VCL – vizuális komponensek könyvtára) illesztünk be új objektumokat.

Vizsgáljuk meg egy kicsit közelebbről, hogy mi történik a kóddal, amikor új komponenseket hozunk létre. Ehhez először kezdjünk egy új projektet, és nyissuk meg a kódszerkesztőt. Megfigyelhető, hogy a form **interface** blokkjának **uses** sora így néz ki:

```
type
  TForm1 = class(TForm)
```

A színfalak mögött a következők történnek:

- A class kulcsszó használatából tudhatjuk, hogy a Delphi egy új objektumot hozott létre, amelynek az őse a TForm objektumtípus.
- Mivel az objektumdeklaráció az **interface** blokkban van, ezért más unitok is hozzáférhetnek ehhez az objektumhoz.

Ezt az alábbihoz hasonló sorok követik:

```
var
  Form1: TForm1;
```

Ez jelzi, hogy a Delphi deklarált egy TForm1 típusú új változót.

Most adjunk a formhoz például egy TButton komponenst. A **uses** sor alatti típusdeklaráció megváltozik:

```
type
  TForm1 = class (TForm)
    Button1: TButton;
```

Szemmel láthatóan a Delphi hozzávette a Button1 mezőt a TForm1 objektumhoz. Végezetül adjunk egy gombot és az ahhoz tartozó OnClick eseménykezelőt a formhoz. A type rész így változik:

```
type
  Tform1 = Class(TForm)
    Button1: Tbutton;
    procedure ButtonClick(Sender: TObject);
```

Nem délibábot látunk, ez tényleg kísértetiesen hasonlít a DeckOfCards objektum definíciójára. Itt láthatjuk, hogy a Delphi mennyire megkönnyíti a dolgunkat, amikor vizuális objektumokkal és azok metódusaival akarunk dolgozni. Miként ez a példa mutatja, maga a form is egy Delphi objektum. Ha még pontosabban akarunk fogalmazni, akkor azt mondhatjuk, hogy a Delphi megírja helyettünk a vizuális objektumok előállításához szükséges kódot (ebben az esetben az ablakot és az azon lévő gombot). Nyilvánvalóan a Delphi objektumok funkciók széles skáláját öröklik a VCL-ből! (Azzal csak jól járunk, hogy a Delphi végzi a deklarációkat, hiszen így a szükséges változtatásokkal nekünk nem kell foglalkoznunk, azok automatikusan végbemennek, amikor komponenseket illesztünk a formra, illetve törlünk onnan. Továbbá a memóriafoglalást is a Delphi végzi a VCL komponensek számára. Így fölösleges aggódnunk a Visual Component Library-ból átvett komponensek memóriaterületének biztosítása és felszabadítása miatt.)

Megjegyzés: Semmiképpen ne tegyük saját objektumdefinícióinkat a form ***type*** *szekciójába. (Objektumváltozók deklarálását azonban megtehetjük itt.) Az itt található információk alapján hozza létre a Delphi a form fájlt (.DFM), bármiféle változtatás megnehezíti (néha lehetetlenné teszi) a Delphi számára, hogy megjelenítse a formot.*

A formdeklaráció private és public részei

Az újonan megnyitott form type szakaszára pillanva azt tapasztaljuk, hogy az hét sorból áll:

```
type
  TForm = class(TForm)
  private
    {Private declaration}
```

```
public
  {Public declaration}
end;
```

Ugyanúgy, mint a felhasználó által készített objektumoknál, itt is a **private** kulcsszó alatt deklarált adatok és metódusok nem lesznek láthatók más unitok számára, míg minden, ami a deklaráció **public** részében szerepel közkincs, azaz más unitokból is elérhető. (Ez azt is jelenti, hogy amíg egy objektumot az őt tartalmazó unitban kívánunk használni, addig nincs különbség a **public** és a **private** kulcsszó hatása között.) A nyilvános módon (a **public** kulcsszóval) deklarált adatokra és metódusokra történő hivatkozáskor, a szintaxisban szerepel a form neve. Az általános szintaxisban pontot teszünk a form és a metódus neve közé:

FormNév.NyilvánosMetódusNév

A Sender paraméter

A könyvben több helyen láthattuk, hogy minden eseménykezelő eljárás rendelkezik egy TObject típusú Sender paraméterrel. Mivel a TObject a Visual Component Library objektumainak közös őse, a Sender paraméter – megfelelő használat esetén – rendkívüli rugalmasságot kölcsönözhet a programoknak. A Sender paraméter segítségével megtudhatjuk, hogy melyik komponens váltotta ki az eseményt. Ez a kulcs ahhoz, hogy ugyanazt az eseménykezelőt több komponens is használhassa. Ha például el szeretnénk érni, hogy egy nyomógomb lenyomásának ugyanaz legyen a hatása, mint egy menüpont választásának, akkor fölösleges lenne kétszer megírnunk ugyanazt az eseménykezelő eljárást. Egy közös eseménykezelőt a következőképpen valósíthatunk meg:

1. Írjuk meg az első objektum eseménykezelő eljárását.
2. Válasszuk ki az új objektumot (objektumokat).
3. Menjünk az objektumfelügyelő Event oldalára.
4. Kattintsunk a nyílra, amely az eseményt tartalmazó sor jobb oldali cellájában található. (A Delphi itt felsorolja a form összes kompatíbilis eseménykezelő eljárását.)
5. A legördülő listában válasszuk ki az eseményt.

Ha az esemény a komponensek bármelyikében bekövetkezik, akkor a Delphi meghívja a közös eseménykezelőt. (A sorrendet meg is cserélhetjük: először kiválasztjuk a komponenseket, és csak utána írjuk meg az általuk közösen használt eseménykezelő eljárást.)

Természetesen fontos, hogy valamilyen módon meg tudjuk különböztetni, hogy melyik komponens hívta meg a közös eseménykezelőt. Ezt a Sender paraméter kiolvasásával tehetjük meg. Az alábbi példa ennek használatát szemlélteti:

```
begin
if Sender = Button1 then
   {Az ide írt kódrészlet csak a Button1 gomb
   lenyomásakor hajtódik végre.}
else
   if Sender = Button2 then
      {Az ide írt kódrészlet csak a Button2 gomb
      lenyomásakor hajtódik végre.}
end;
```

A Sender paramétert általában arra használjuk, hogy megtudjuk a hívó komponens nevét vagy – az **as** operátor segítségével – annak típusát. (Az **as** operátor használatáról többet a beépített helpben találhatunk.)

Objektumváltozók

Az objektum-változók olyan változók, amelyek vagy objektum, vagy objektumokból álló strukturált típusúak. Például:

```
var
  Form: TForms;
  AnEditBox: TEdit;
  ArrayOfEditBox: array[1..10] of TEdit;
```

A Delphiben használhatunk objektumelemű tömböket és rekordokat is.

Az objektumváltozók nélkülözhetetlen eszközt jelentenek hatékony Delphi projektek írásához. Egyik fontos alkalmazási területük, amikor közös eseménykezelő eljárások használatánál meg akarjuk tudni, hogy a

Sender paraméter milyen objektum. Egy TEdit, amelynek nincs Caption tulajdonsága, vagy egy TLabel, amelyiknek van? Ennek eldöntésére az **is** operátort használjuk.

Adjunk például formunkhoz egy Edit1 nevű TEdit és egy Label1 nevű TLabel komponenst. Majd hozzunk létre egy közös OnClick eseménykezelő eljárást, amely az alábbi kódot tartalmazza:

```
procedure TForm1.Edit1Click(Sender: TObject);
  if Sender is TEdit then
TEdit(Sender).Text := 'A szerkesztési ablakra
kattintott';
  else
 TLabel(Sender).Caption := 'A címkére kattintott';
end;
```

Az eljáráson belül a Sender egy objektumváltozó. Azt, hogy a Sender egy TEdit, TButton, TLabel vagy más komponens-e, az **is** operátorral dönthetjük el.

Mivel a Sender paraméter maga TObject típusú, ezért közvetlenül nem vizsgálhatunk meg olyan dolgokat, mint a hívó komponens Name tulajdonsága. A Sender.Name-nek nem lenne értelme, hiszen a TObject osztálynak nincs Name tulajdonsága – egy TObject példány bármilyen Delphi objektum lehet. Ekkor egy speciális eljárással (az eljárás hétköznapi értelmében), az úgynevezett objektumkonverzióval (casting) érhetünk célt. Ilyen módon egy bizonyos típusú objektumot egy más típusú objektummá alakíthatunk (hasonlóan a típuskonverziókhoz), de természetesen ezt csak kompatíbilis objektumok esetén végezhetjük el. Nem konvertálhatunk például egy szerkesztési ablakot memóablak-komponenssé.

Tegyük fel, hogy van egy sor szerkesztési ablakunk, amelyekhez tartozik egy közös eseménykezelő eljárás és ebben az eseménykezelőben el szeretnénk dönteni, hogy a hívó komponens üres-e.

```
procedure TForm1.Edit1Click(Sender: TObject);
var
  EditBox: TEdit;
begin
  EditBox := (Sender as Tedit);{az 'as' operátorral
  végezzük a másolást}
```

```
  if EditBox,Text <> '' then
  ShowMessage(EditBox.Name);
end;
```

Megjegyzés: Az **as** *kulcsszóval ideiglenes konverziót hajthatunk végre. Lehetőségünk van azonban tartós konverziót is végeznünk Delphi objektumokkal. Ehhez az osztálynevet használjuk függvényként, amelynek paramétere az objektumváltozó (általában TObject típusú). Általában kívánatosabb az* **as** *kulcsszóval végzett másolás.*

Hasonlóan ahhoz, ahogy az eseménykezelő eljárások használnak objektumparamétereket, mi is gyárthatunk olyan alprogramokat, amelyek paraméterként (ez lehet egy vagy több) objektumváltozókat használnak. Példaként tekintsünk egy olyan eljárást, amely a hívó szerkesztési ablak tartalmát csupa nagybetűssé alakítja:

```
procedure UppercaseTextInEditBox(EditBox: TEdit);
begin
  EditBox.Text = UpperCase(EditBox.Text);
end;
```

Objektumváltozókat használunk akkor is, amikor valamilyen művelet (pl. keresés) részeként végig kell mennünk a form összes komponensén. Példaként írjunk egy olyan eljárást, amely végignézi a form összes komponensét, és ha szerkesztési ablakot talál, annak tartalmát letörli. Ehhez szükségünk lesz a form Components és ComponentCount tulajdonságaira, amelyek segítségével hozzáférhetünk a form komponenseihez. Az eljárás egy lehetséges megvalósítása:

```
procedure BlankEditBoxes(FormName: TForm);
var
  i: Integer;
begin
  for i := 1 to FormName.ComponentCount - 1 do
    if FormName.Components[i] is TEdit then
      (FormName.Components[i] as TEdit).Text = '';
end;
```

Megjegyzés: A Delphinek van egy Controls nevű tulajdonsága is. A Components és a Controls tulajdonság között az a különbség, hogy amíg az előbbi az objektum összes komponensét felsorolja, addig az utóbbi az objektum gyermekablakait sorolja csak fel. Ez akkor válik fontossá, amikor egy tartalmazó komponenssel, például egy csoportablakkal van dolgunk. A csoportablak komponenseinek a form a tulajdonosa, így a form Components tulajdonsága tartalmazza őket. Ezek egyben a csoportablak gyermekablakai, tehát nincsenek benne a form Control tulajdonságában, megtalálhatóak viszont a csoportablak Controls tulajdonságában.

■ *Delphi objektumok létrehozása futási idő alatt*

Elég gyakran felmerül az igény arra, hogy futási idő alatt hozzunk létre Delphi objektumokat. Szükségünk lehet egy új sztringlistára például adatok tárolására vagy előfordulhat – valamilyen speciális feladatban –, hogy új szerkesztési ablakot kell nyitnunk a formon. Még az is elképzelhető, hogy egy új formot kell előállítanunk. (Természetesen ezeket fejlesztési időben is létrehozhatjuk, és elrejtve tarthatjuk egészen addig, amíg szükségünk nincs rá. Ennek gyakori alkalmazása azonban, túl azon, hogy memóriaigényes, lassíthatja a projekt futását, és felemésztheti a Windows szűkös erőforrásait.)

Tegyük fel, hogy egy sztringlistát akarunk előállítani futási idő alatt. A sztringlistákat a Delphi TStrings osztályából származtatjuk. A szükséges kód alább látható:

```
var
  MyStringList : TSringList;
begin
  MyStringList := TStringList.Create;
  try
    {a sztringlistához kapcsolódó kód}
  finally
    MyStringList.Free;
  end;
end;
```

(A TStringList a TStrings leszármazottja, amit akkor használunk, amikor a listát manipulálnunk kell.)

Megjegyzés: Minden objektum létrehozásához szabad memóriát kell biztosítanunk, ezért a kódot a ***try/finally*** *blokkba kell elhelyeznünk. (l. még a 8. fejezetet.)*

Elképzelhető, hogy egy olyan sztringlistára lesz szükségünk, amely elérhető az alkalmazás futásának teljes ideje alatt. Ekkor a következőket kell tennünk:

1. Deklaráljunk egy TStringList típusú mezőt a form deklarációban.
2. A sztringlistát a form OnCreate eseménykezelőjében hozzuk létre.

Megjegyzés: Mindig szabadítsuk fel a sztringlistáknak lefoglalt memóriaterületet. Általában ezt az OnDestroy eseménykezelő eljárásban hajthatjuk végre.

Új vizuális objektumok létrehozása futási időben

Tegyük fel, hogy futási időben egy szerkesztési ablakot akarunk hozzáadni a formhoz. Az ezt megvalósító kód valahogy így nézne ki:

```
var
  NewEditBox: TEdit;
begin
  NewEditBox := TEdit.Create(Self); {az objektum
  most már létezik!}
  NewEditBox.Parent := Self; {Self jobb, mint
  a Form - főleg az OnCreate eseményben}
end;
```

Megjegyzés: A TEdit.Create(Self) sor után az objektum létrejött, ekkor kell beállítanunk a különböző komponenseket, úgymint a Left, Top, Visible stb. A tulajdonságoknak a Parent beállítása előtt adjunk értéket, mert így a változások nem látszanak a képernyőn.

A következő néhány bekezdésben a példában használt kulcsszavak és metódusok rövid leírását közöljük.

Self

Az általunk létrehozott komponensek tulajdonosa egy másik komponens kell hogy legyen. A Create metódus általános szintaxisában ez így tükröződik:

Create(Tulajdonos: TComponent)

A legtöbb esetben a **self** kulcsszót használjuk a Create paramétereként. Ez azt jelenti, hogy a komponensnek az az osztály lesz a tulajdonosa, amelyben a metódus található – amely általában egy form. Másrészt ha egy másik komponenst jelölünk meg az új komponens tulajdonosaként, akkor a továbbiakban az a komponens felel az objektum által elfoglalt memória felszabadításáért.

Megjegyzés: *Ezen saját maguk alá rendelt objektumok esetén is – csakúgy mint a nemvizuális objektumoknál – szükség van a memória felszabadítására. (Használjuk a korábban tárgyalt Free metódust.)*

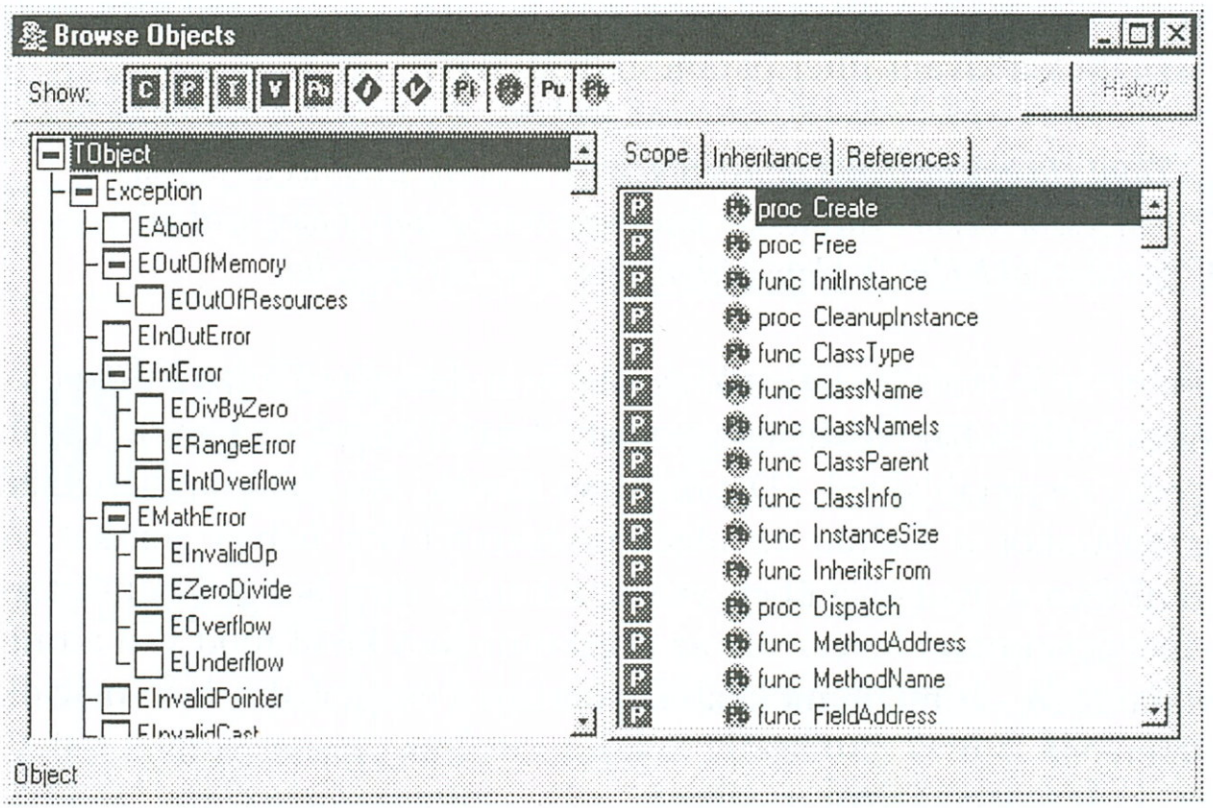
A Parent tulajdonság

A Parent tulajdonság értéke a szülő komponens (pl. TForm, TPanel vagy TGroupBox) nevét tartalmazza.

Visual Basic tipp: *Ha a Visual Basic vezérlőelem-tömbjéhez (control array) hasonló struktúrát kívánunk létrehozni, akkor deklaráljunk egy objektumváltozókból álló tömböt, majd hozzuk létre egyesével az objektumváltozókat a fent ismertetett technikákkal.*

■ *A Delphi objektumböngészője*

Vannak helyzetek, amikor hasznos lehet a Delphi objektumok hierarchiájának ismerete. A View | Browser menüponton keresztül elérhető objektumböngészőben (ObjectBrowser) lehetőségünk van az osztály-hierarchia tanulmányozására. Az objektumböngésző azonban ennél jóval többre is alkalmas. Segítségével elemezhetjük projektünk unitjait és globális változóit. Továbbá gyors hozzáférést biztosít a program olyan fontos részeihez, mint például a globális változók deklarációja. A Delphi 32 objektumböngészőjét a 7-2. ábra mutatja. (A Delphi 16-é pontosan így néz ki, leszámítva a Windows 95 címsor elemeit.)



7-2. ábra. Az objektumböngésző ablak

Megjegyzés: *Az objektumböngésző használatához legalább egyszer le kell fordítanunk a programot. Ahhoz, hogy a böngészővel dolgozzunk, nem szükséges, hogy* break *módban legyünk.*

Megjegyzés: *Mint a legtöbb beépített Delphi eszköznek, az objektumböngészőnek is van saját gyorsmenüje, amelyen keresztül elérhetők a böngésző legfontosabb részei.*

Mivel az objektumböngészőn keresztül a Delphi belső működésébe tekinthetünk be, ezért nem könnyű kiigazodni az ott látható információáradatban. A böngészőt általában ritkán használjuk, de amikor mégis szükségünk van rá, érdemes kiszűrni a számunkra érdektelen információkat. A szűrést az objektumböngésző tetején található Show paletta ikonjain keresztül szabályozhatjuk. A szűrők (vagy filterek) beállítására három különböző módszer áll rendelkezésünkre:

- Nyomjuk le a kívánt szűrő gombját. (Például a C gombot, ha a konstansokat, a P gombot, ha a függvényeket és eljárásokat kívánjuk tanulmányozni.)
- Az Environment Options dialógusablak Browser oldalán a jelölőnégyzetek segítségével válasszuk ki (vagy iktassuk ki) a kívánt filtereket.
- Használjuk a kívánt filter parancsbillentyűjét.

Tipp: *Ha egy bizonyos típusú objektum miatt használjuk az objektumböngészőt, akkor érdemes az Environment Options dialógusablak Browser oldalán ezt alapbeállításként (default) megadni.*

Miként az a 7-2. ábrán látható, az objektumböngésző két (átméretezhető) részből áll. A bal oldalit felügyelőablaknak (Inspector pane), a jobb oldalit pedig részletablaknak (Detail pane) hívják. Az objektumböngésző első használatakor a Delphi az objektumhierarchiára helyez minket. A legegyszerűbben a gyorsmenüben változtathatjuk meg böngészésünk tárgyát. A menüpontokon keresztülz az objektumokon kívül mód van a unitokra vonatkozó, szimbólumokkal kapcsolatos és globális információk böngészésére is.

Tipp: *Az objektumböngészőnek van egy okos keresőfunkciója, amely megkönnyíti a tájékozódást a böngészőben. A keresés az egymás után leütött betűk alapján halad a böngészőben attól függően, hogy hol talál az addig begépelteknek megfelelő pontot. A begépelt karaktereket a Show paletta szűrőgombjai és a History gomb között láthatjuk. (Az ESC gomb lenyomásával tiszta lappal indulhatunk a keresésben, míg a CTRL-N a következő találatra ugrik.)*

A fordítási direktívák és hatásuk az objektumböngészőre

Az objektumböngészőt csak a már lefordított kódra alkalmazhatjuk. A projekt lefordításának módja befolyásolja a böngészőben látottakat. Amennyiben az objektumböngészőt alkalmassá kívánjuk tenni hibakeresésre, akkor meg kell változtatnunk néhány fordítási direktívát. (Enélkül kizárólag a unitok **Interface** blokkjában deklarált szimbólumokhoz férhetünk hozzá.) A 7-1. táblázat foglalja össze a lehetőségeket.

7-1. táblázat. Fordítási direktívák hatása az objektumböngészőre

Direktíva	*Hatása az objektumböngészőben*
Debugging Information	A modul Implementációs részében definiált (nyomkövetési információk) szimbólumok hozzáférhetőségét biztosítja.
Local Symbols	A részletablak References oldalán (lokális szimbólumok információi) megjeleníti a programsor-szám hivatkozásokat
Symbol Info	Egy adott függvényen vagy eljáráson belül szimbólum-referencia információk deklarált azonosítók böngészését teszi lehetővé.

Tipp: *Ezen fodítási direktívák bekapcsolt állapotban a lefordított program futását lassítják, méretét pedig megnövelik, ezért a végső fordítás előtt feltétlenül kapcsoljuk ki őket.*

A felügyelőablak

Amikor objektumokat vizsgálunk a felügyelőablakban, akkor azok öröklődési fáját láthatjuk, amikor unitokat vizsgálunk benne, akkor azok teljes listáját tanulmányozhatjuk. (Ez magában foglalja azokat a unitokat, amelyeket mi adtunk a programhoz, és azokat is, amelyeket a Delphi szerkesztett hozzá, mint pl. a System unitot.) Az objektumok származását a grafikus megjelenítési mód teszi követhetőbbé. (A 7-2. ábrából is leolvasható, hogy a közös ős a TObject). A fát kibonthatjuk ill. összehúzhatjuk, ha a nevek bal oldalán található kis négyzetre kattintunk. A közös őstől származó objektumok az ősből kiinduló ágból ágaznak el. A négyzetben lévő pluszjel utal arra, hogy az ág kibontható, a mínuszjel pedig arra, hogy már kibontott állapotban van.

Tipp: Az objektumböngésző kezelését sok gyorsítóbillentyű segíti, ha a böngészőt sokat használjuk, érdemes ezek egy részét fejben tartani. A parancsbillentyűk a beépített help „ObjectBrowser Keyboard Shortcuts" pontban vannak felsorolva.

Ha a felügyelőablakban kijelölünk egy objektumot, akkor a Delphi a részletablakot felfrissíti.

A részletablak

Ennek az ablaknak három oldala van: Scope (hatáskör), Inheritance (öröklődés) és References (hivatkozások). A Scope oldal azon objektum szimbólumainak listáját tartalmazza, amelyet a felügyelőablakban kijelöltünk. Az Inheritance oldalon a kijelölt objektum öröklődési fáját láthatjuk, a References oldalon a szimbólum kódbeli előfordulásának megfelelő fájlnevek és programsor-számok szerepelnek.

Objektumok mélyrehatóbb vizsgálata

Ha egyes objektumokkal kapcsolatos további információkra is kíváncsiak vagyunk, akkor kattintsunk kettőt az objektumra a felügyelőablakban. Ekkor a részletablak tartalma áttevődik a felügyelőablakba és a részletablakban a kiválasztott objektum deklarációi lesznek láthatók. Ha az objektum kódjára is kíváncsiak vagyunk, akkor a CTRL-ENTER billentyűkombináció használatával a fókuszt a kódszerkesztő ablak kapja meg, és a Delphi a kurzort a kód azon részére állítja, ahol az objektumot deklaráltuk. Általában elmondható, hogy amikor függvényt vagy eljárást jelölünk ki, akkor a Delphi az ahhoz tartozó kódot jeleníti meg; amikor konstanst, tulajdonságot, változót vagy típust, akkor a deklarációs részt láthatjuk a kódszerkesztő ablakban.

8. FEJEZET

Hiba- és kivételkezelés

Akármennyire is ügyelünk a hibák kiküszöbölésére, lehetetlenség megelőzni a mindennapos és hajmeresztő malőröket, amilyeneket egy tapasztalatlan felhasználó képes okozni. Egy olyan program, amelyik felborul vagy lefagy, amikor a printerből nyomtatás közben kifogy a papír vagy a felhasználó megpróbál megnyitni egy nem létező fájlt, nem lesz nagyon népszerű. Egy nehezen összezavarható, stabil program létrehozásánál törekedni kell a végzetes hibák kikerülésére (néha tudni kell büszkén leereszkedni). A Delphi mindkét változata támogatja az ún. védett (protected) blokkokat. Ez a megoldás a C++-ból került át az Object Pascalba és az idejétmúlt On Error Goto-t hivatott helyettesíteni, amelyet a Visual Basic is használ. A védett blokkok használata biztosítja, hogy a kódblokkban keletkezett hibát a program strukturált módon kezelhesse.

Amikor egy sor végrehajtása során hiba keletkezik, a Delphi létrehoz egy a TObjectből származtatott objektumot, amelyet *kivétel objektumnak* hívunk. Miként azt ebben a fejezetben még látni fogjuk, ezen objektum tulajdonságainak elemzésével határozhatjuk meg a hiba okát. (Így a vezérlést visszaadhatjuk a hibát kiváltó kódrészletnek – a hiba újbóli előidézésének kizárásával. Például megkérhetjük a felhasználót arra, hogy tegyen papírt a nyomtatóba.)

Megjegyzés: *Amikor kivételkezelést tartalmazó alkalmazásokat fejlesztünk, nem mindig előnyös, hogy a Delphi megszakítja a programot a kivételek fellépésekor. Ez ugyan hasznos lehet, ha azt akarjuk, hogy a Delphi megmutassa a kivétel fellépésének a helyét, de zavaró, ha éppen a program kivételkezelését szeretnénk tesztelni. Ezt az alapbeállítást a Delphi 32-ben a Tools | Options pont választása után, a Delphi 16-*

ban az Options | Environment pont választása után, a megjelenő dialógusablak Preferences oldalán változtathatjuk meg, a Break on Exceptions opció kikapcsolásával.

■ *Védett blokkok*

A kivétel- és hibakezelést egy adott kódblokkban úgy aktivizálhatjuk, ha a blokkot a **try/except** kulcsszavak közé helyezzük. Az általános sablon ennek használatára így néz ki:

```
try
    {utasítások listája}
except
    on...do  {az egyes kivételek kezelése};
else {az összes többi kivétel kezelése};
end;
```

Nem kívánatos, hogy a végrehajtás véletlenül belekerüljön a hibakezelő részbe. A **try/except** blokk biztosítja, hogy a hibakezelés elkülönüljön a program „normális" menetétől, ami által a program is áttekinthetőbbé válik.

Alkalmanként szükséges, hogy a Delphi végrehajtson bizonyos programrészletet függetlenül attól, hogy melyik a védett blokk. (Ez elég gyakori azon részeknél, ahol memória vagy Windows-erőforrás foglalása történik, ugyanis ezeket vissza szeretnénk kapni.) Ilyen esetekben a **finally** kulcsszóval különítjük el a mindekképpen végrehajtandó blokkot:

```
    {az erőforrás-foglalást végző kód helye}
try
    {memóriafoglalásnak a try...finally blokkot
     megelőzően meg kell történnie, mert elképzel-
     hető hogy az hibával zárul}  try
    {utasítások helye}
    except
      on .. do {az egyes kivételek kezelése};
      else {az összes többi kivétel kezelése};
    end;
```

```
finally
    {erőforrások felszabadítását végző kód}
end;
```

Itt a **finally** kulcsszó utáni blokk, amely az erőforrás felszabadításáról gondoskodik, mindenképpen végrehajtódik.

Többszintű kivételkezelést is végezhetünk azáltal, hogy több try/except ill. try/finallly blokkot ágyazunk egymásba. Ha az egyik blokk nem kezel egy adott kivételt, akkor a Delphi eggyel kijjebbi szinten lévő kivételkezelő blokkra ugrik, és ezt addig ismétli, amíg nem talál egy védett blokkot (azaz egy olyat, ahol a kivételt fogadják). Tegyük fel például, hogy egy beágyazott for-next ciklus külső részében van a kivételkezelő rész, és belső, nem védett ciklusban egy kivétel lép fel. Mivel itt nincs kivételkezelő, ezért a Delphi a kivételt a külső ciklus kivételkezelőjének adja át.

Amennyiben alkalmazásunk nem kezeli a kivételt okozó hibát, akkor a Delphi az ő alapértelmezés szerinti kivételkezelőjét fogja használni, amely megjelenít egy üzenetablakot, benne az általa vélt hibafajtával, majd megpróbálja folytatni a programot. Ha nem tudja, akkor a program lefagy. Amint viszont elkezdünk kivételt kezelni (hibákat elfogni) – a try/except blokk segítségével –, a kivételek nem zavarják meg a program zökkenőmentes működését.

Tipp: *Mivel igen ritkán hagyatkozunk egyedül a Delphi saját kivételkezelőjére, érdemes a TApplication objektum OnException eseménykezelőjébe saját kódot írni. Ez minden kivétel fellépésekor végrehajtódik – mielőtt az alapértelmezett eseménykezelő kapná meg a vezérlést.*

Minden esetre a try/finally blokkban kell hogy legyen egy olyan kód, amely azonosítja a problémát, és ha lehetséges, rendbehozza. Ha a hibát nem lehet kijavítani, akkor a Delphi végrehajtja az **except** blokk egyéb hibakezelésre létrehozott **else** részét.

Visual Basic tipp: *A Delphi nem adja vissza a vezérlést a kivételt okozó programsor utáni utasításnak, hanem a végrehajtás a megfelelő* ***on...do*** *utasítással folytatódik.*

A hibát nem javíthatjuk ki addig, amíg nem tudjuk, hogy az miért történt. A problémát a kivételkezelő **on...do** utasításával azonosíthatjuk. Ennek segítségével kezelhetjük az egyes kivételeket. Például, ha egy általunk definiált függvényben az

```
except
on EDivByZero do Result := 0;
```

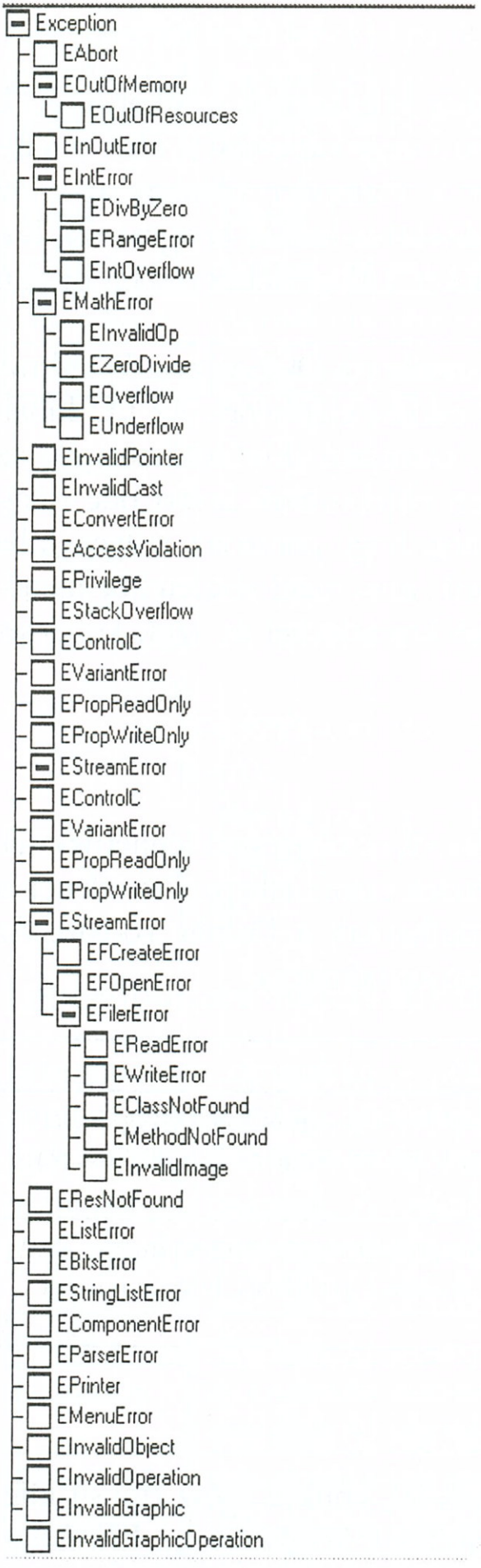
szerepel, akkor a hiba, amelyet nullával való osztás idézett elő, azt eredményezi, hogy a függvény értéke nulla lesz.

■ *A kivételhierarchia*

A 8-1. ábra a Delphi 32 kivételhierarchiáját mutatja (lényegében megegyezik a Delphi 16-éval). Látható, hogy minden kivétel az Exception objektumból származik. Sokaknak közülük – mint pl. az EIntError-nak (l. később) – vannak specializáltabb hibaállapotokat leíró leszármazottaik (alosztályaik). A könyvben a kivételek teljes ismertetésére, azok nagy számára való tekintettel (több tucat) nem kerülhet sor. Az itt nem tárgyalt kivételekről a beépített súgóban kaphatunk felvilágosítást.

Tipp: *A Delphi a futási idejű könyvtárhoz (Runtime library) kapcsolódó hibák széles skáláját tudja azonosítani. A beépített help-ben erről az „exceptions" szó alatt, a „Handling RTL Exceptions" pontban kaphatunk bővebb információkat. A rövid magyarázatokhoz is fordulhatunk egy-egy speciális hiba okának kiderítésére.*

A komponensek is támogatják a hibakezelést, probléma esetén azok is kivételeket váltanak ki – a kihívást ezen hibák azonosítása jelenti. A legtöbb komponensben történő kivétel programozási hiba folytán keletkezik, úgymint egy tömb nem létező indexére történő hivatkozás. A most következő néhány oldalon rövid ízelítőt adunk a futási idejű könyvtár rutinjaihoz kapcsolódó leggyakoribb hibákból.



8-1. ábra. A kivételhierarchia

Input/output kivételek

Input/output (I/O) kivételek általában akkor keletkeznek, amikor az RTL (Runtime library) egy fájlt próbál megnyitni vagy kapcsolatot akar teremteni egy I/O eszközzel. Fontos, hogy a Compiler oldalon az „I/O checking" opció be legyen kacsolva (I+ direktíva hatása is ugyanez), különben a Delphi váltja ki ezeket a kivételeket. (Alapértelmezésben be van kapcsolva.)

A legtöbb I/O kivétel akkor keletkezik, amikor a DOS vagy a Windows problémákba ütközik egy fájl vagy külső eszköz elérésénél. Az I/O hibák kezeléséért az Exception objektumból származó EInOutError és az EStreamError alosztályai a felelősek. Ezen objektumok ErrorCode tulajdonságának elemzésével határozhatjuk meg, hogy pontosan milyen hiba történt. (Példát a 10. fejezetben találhatunk, ahol majd részletekbe menően foglalkozunk a fájlkezelésnél keletkező kivételekkel és azok kezelésével.)

Heap kivételek

Heap kivételek dinamikus memóriafoglalásnál és címzésnél léphetnek fel. A szabad memória hiányából adódó kivételek szinte elképzelhetetlenek a Delphi 32 bites változatában, de előfordulhatnak a Delphi 16-ban. Az alábbi táblázat a lehetséges heap kivételeket tartalmazza.

Kivétel	*Jelentés*
EOutOfMemory	Nem volt elég szabad hely a heap-ben a kívánt művelet elvégzésére. Az EOutOfMemory-nak van egy leszármazottja: az EOutOfResources, amely szintén kiváltható.
EInvalidPointer	Az alkalmazás egy olyan mutatót próbált megszüntetni, amely a heap-en kívülre mutatott. Ez általában azt jelenti, hogy a mutatót már megszüntettük.

Egész aritmetika kivételek

Egész aritmetika kivételek akkor léphetnek fel, amikor egész típusú kifejezésekkel végzünk műveleteket. Ezen kivételek közös őse az EIntError, amelyet általában nem használ a Runtime library, de ebben gyökerezik a többi speciális egész aritmetikai kivétel. Kivételkezelőkben való használa-

tára természetesen lehetőségünk van, ha nem vagyunk kíváncsiak az egész művelet elvégzésekor keletkező hiba konkrét okára. Az egyes kivételeket az alábbi táblázatban foglaltuk össze:

Kivétel	*Jelentés*
EDivByZero	Nullával való osztás történt.
ERangeError	Az értékhatár-ellenőrzés hibát talált.
EIntOverflow	Az egész számokkal végzett művelet túlcsordulást eredményezett.

Példa:

```
begin
  try
    szám := A div B; {egész osztás}
  except
    on EDivByZero do
       begin
         MessageDlg( 'Nullával osztott!', mtError,
         [mbOK], 0);
       end;
```

Lebegőpontos matematikai kivételek

Lebegőpontos típusokkal – például Single – végzett műveletek lebegőpontos matematikai kivételekhez vezethetnek. Ezek közös őse az EMathError. Ebből is, mint az EIntError-ból, a specializáltabb kivételosztályok származnak. Az alábbi táblázat az EMathError családjába tartozó lebegőpontos matematikai kivételeket foglalja össze:

Kivétel	*Jelentés*
EInvalidOp	Egy érvénytelen matematikai művelet (például gyökvonás negatív számból) került végrehajtásra.
EZeroDivide	Nullával való osztás történt.
EOverflow	A lebegőpontos művelet túlcsorduláshoz vezetett.
EUnderflow	A lebegőpontos művelet alulcsorduláshoz vezetett.

Objektumkonverziós kivételek

Az objektumkonverziós kivételek akkor lépnek fel, amikor az as operátor bal oldalán megadott objektum osztálya nem egyezik meg a jobb oldalán megadott objektum osztályával. Csak egy kivételosztály tartozik ide: az EInvalidCast.

Konverziós kivételek

A konverziós kivételek adatkonverziók során léphetnek fel, amikor adatokat kívánunk egyik típusból egy másikba alakítani az erre szolgáló konverziós függvényekkel (pl. IntToStr, StrToInt, StrToFloat stb.). Az egyedüli konverziós kivételosztály az EConvertError.

Hardver kivételek a Delphi 16-ban

Hardver kivételek a Delphi 16-ban két szituációban történhetnek: vagy a processzor egy olyan hibát talál, amit nem tud kezelni, vagy az alkalmazás szándékosan meghív egy megszakítást a végrehajtás felfüggesztése céljából. Hardver hibakezelés nem fordítódik le DLL-ek készítésekor – csak önmagukban futtatható alkalmazások esetén. A hardver kivételek közös őse az EProcessorException, ezt azonban soha nem váltja ki a futási idejű könyvtár. Ebből az osztályból származnak a specializáltabb hardver kivételek. Az egyes hardver kivételeket az alábbi táblázat gyűjti egybe.

Kivétel	*Jelentés*
EFault	Az EProcessorException leszármazottja. A processzorhiba kivételek őse.
EGPFault	Általános védelmi hiba kivétel, általában inicializálatlan mutató vagy objektum okozza.
EStackFault	A program illegálisan próbált hozzáférni a processzor veremszegmenséhez.
EPageFault	A Windows memóriakezelője valamilyen oknál fogva nem tudja helyesen használni a swap-fájlt.
EInvalidOpCode	A processzorhoz egy érvénytelen utasítás továbbítódik. Ez általában azt jelenti, hogy a processzor adatokat kapott utasítások helyett vagy inicalizálatlan memóriaterületre tévedt.
EBreakPoint	Az alkalmazás egy *breakpoint* (töréspont) megszakítást generált.
ESingleStep	Az alkalmazás egy *single-step* megszakítást generált.

Megjegyzés: A táblázatban található kivételek csak a Delphi 16-ban alkalmazhatók, a Delphi 32-ben új hardver kivételek vannak: EAccessViolation, EPrivilege és EStackOverflow, amelyek mind az Exception osztály leszármazottjai.

Kivételosztályok kezelése

A fejezet elején kiderült, hogy bármilyen utasításblokkot körülvehetünk kivételkezelőkkel, amelyeket esetleg egymásba is ágyazhatunk. A kivételkezelést az egyes kivétel osztályokhoz kötjük. Például:

```
try
  { egész aritmetikai műveletek }
except
  on ERangeError do { értékhatárhibák kezelése };
  on EIntError do { más egész aritmetikai hibák
  kezelése };
end;
```

Ez a kód alkalmas minden egész artitmetikai hiba (beleértve az értékhatárhibákat) kezelésére. Érdemes megjegyezni, hogy ha a fenti példában az EIntError kivételkezelője megelőzte volna az ERangeError kezelőjét, akkor az utóbbi sohasem kaphatta volna meg a végrehajtást.

Kivételelek ismételt előidézése

Előfordulhat, hogy egy védett blokkban történő kivételkezelés után a kivételt továbbítani kívánjuk a külső kivételkezelőnek. Ilyen esetekben problémát okoz az, hogy a helyi kivételkezelés után a kivételobjektum megsemmisül, így a külső blokk kivételkezelő részéhez már nem juthat el. Módunk van azonban a kivétel életben tartására a belső kivételkezelés megtörténte után is, így mégiscsak aktivizálhatjuk a külső blokkban a megfelelő on...do utasítást. Ehhez a **raise** kulcsszót kell használnunk.

A gyakorlatban akkor használhatunk például ilyen kettős hibaellenőrzést, amikor a belső kivételkezelő csak valamilyen üzenet továbbítását végzi a felhasználóhoz, a probléma orvoslását a külső kezelőre bízza. A megoldás váza így néz ki:

```
try
  { külső utasítások }
  try
    { belső utasítások }
  except
    on EValamilyenHiba do
    begin
      { a belső utasítások kivételkezelése }
      raise;    {ez váltja ki a kivételt }
    aut;
  end;
except
  on EValamilyenHiba do ...; { általános hibakezelés
}
end;
```

Ha az első („belső utasítások") rész kivételt hoz létre, akkor a Delphi csak a külső rész kivételkezelőjét („általános hibakezelés") hajtja végre. Amennyiben a „belső utasítások" blokk vált ki egy kivételt, akkor természetesen a Delphi végrehajtja a belső kivételkezelő részt, de *a **raise** kulcsszó miatt a külső kivételkezelő is végrehajtódik.*

***Tipp:** Kivételek ismételt előidézésével speciális esetekben egyszerűen biztosíthatunk hibakezelést a meglévő kivételkezelő részek elvesztése (vagy lemásolása) nélkül.*

■ *Védett erőforrásfoglalások*

Minden jól működő alkalmazás kell hogy tartalmazzon olyan hibakezelést, amellyel hiba esetén is biztosított a rendszer erőforrásainak felszabadítása.

Megjegyzés: Minden alkalommal, amikor mutatóknak memóriát foglalunk (például a GetMem vagy a New használatával) vagy létrehozunk egy objektumot, vagy fájlokat nyitunk meg, akkor értékes Windows erőforrásokat használunk el.

Ha az alkalmazásunk egy súlyos hibát detektál, ami a programot futásának befejezésére és az alkalmazás teljes bezárására kényszeríti, akkor a „végső búcsú" előtt nem árt elvégezni a szükséges memóriafelszabadításokat és a logikai adatállományok bezárását. (Ellenkező esetben a felhasználó szomorúan tapasztalhatja, hogy a Windows erőforrások csökkenésével csökken rendszerünk teljesítőképessége is, ami extrém esetben a rendszer újraindítását teszi szükségessé.)

Az alábbi példaprogram memóriát foglal egy mutató és egy sztringlista számára, elfog egy *nullával való osztás* hibát, majd felszabadítja az előzetesen lefoglalt memóriát.

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  Aptr: PChar;
  AnInt, BadToDivide: Integer;
  MyStringList: TStringList;
begin
  BadToDivide := 0;
  MyStringList := TStringList.Create;
  GetMem(APtr, 50000);    {50 000 byte memóriát
                          foglal le}

{Ha a GetMem sikertelen (EOutOfMemory kivétel lép
fel), akkor a MyStringList objektum nem szabadul
fel.}
  try
  AnInt := 10 div BadToDivide;  { ez hibához vezet }
  finally
    FreeMem(Aptr, 500000);    {ez a kód mindig végre-
                              hajtódik}
    MyStringList.Free;
end;
```

■ *Felhasználó által definiált kivételek*

A Delphi előre definiált osztályaiból a programozó saját osztályokat származtathat, nincs ez másként a kivételek esetében sem. Az Exception osztályhoz kívánság szerinti alosztályokat adhatunk. Mivel ezek az Exception osztály leszármazottai lesznek, ezeket is használhatjuk a kivételkezelőkben, mint bármilyen más, előre definiált alosztályát az Exception-nek. A szintaxis sem különbözik a futási idejű könyvtár kivételei által használttól.

Két dologról kell gondoskodnunk saját kivételeink használatakor: deklarálnunk kell, azután pedig ki kell váltani őket.

Kivételobjektumok deklarálása

A kivételek olyanok, mint a többi Delphi objektum, így azokat is deklarálnunk kell használat előtt. A standard kivételkezelők csak a Exception objektum típusú kivételeket kezelik. Például:

```
type
  ESajátHiba = class (Exception);
```

Ha most kiváltjuk az ESajátHiba kivételt anélkül, hogy egyedi kivételkezelést írtunk volna számára, a Delphi standard kivételkezelője akkor is fogadja. A Delphi standard kivételkezelője nem csinál mást, mint megjelenít egy üzenetablakot, a kivételnek megfelelő szöveggel.

A saját kivételek létrehozásának előnye az, hogy kiváltásával a program többi részének globális információval szolgálhatunk. A saját kivételeinket a **raise** kulcsszóval válthatjuk ki. A példa ezt illusztrálja:

```
type
  ESajátHiba = class(Exception);
...
if Valami = ValamiMás then
   raise ESajátHiba.Create('Ez az én kivételem');
```

Amikor a Delphi befejezi egy kivétel feldolgozását, akkor a kivétel-példány automatikusan törlődik. Ez azt jelenti, hogy nekünk nem kell törődni a lefoglalt memória felszabadításával.

Megjegyzés: A System unit tartalmaz egy ErrorAddr nevű változót, amely (ahogy azt az angol neve elárulja) azt a memóriacímet tartalmazza, ahol az alkalmazásunkban a kivétel keletkezett. Az ExceptionObject függvény pedig egy olyan mutatót hoz létre, amely a kiváltott kivétel-objektumra mutat.

Csendes kivételek

Ahogy azt korábban láthattuk, a Delphi minden kivételt, amelyhez nem írunk külön kódot, standard kivételkezelőjével fog el, ami egy üzenetablak megjelenését eredményezi. Alkalmanként (például egy kezdő felhasználóknak szánt program esetében) szükségünk lehet arra, hogy olyan kivételeket hozzunk létre, amelyek nem jelenítenek meg üzenetet a képernyőn. Ezeket hívjuk *csendes* kivételeknek. Az Abort eljárás használatával érhetünk legkönnyebben célt. Az Abort automatikusan kiváltja az EAbort kivételt (természetesen az EAbort-ot közvetlenül is kiválthatjuk).

Megjegyzés: Az EAbort-ot leszámítva minden más kivételtípus megjelenít üzenetablakot.

Az Abort eljárás arra kényszeríti a Delphit, hogy hagyja el az aktuális utasításblokkot, de nem függeszti fel a program végrehajtását. (Ez egy másik módja az utasításblokk elhagyásának – a Goto az egyik.)

Megjegyzés: Az EAbort kivételt (Abort eljárást) gyakran használják adatbázis-programozásban. Ha valamilyen oknál fogva például fel kell függeszteni a rekordok módosításának foganatosítását az adattáblában, akkor az Abort eljárást használjuk.

9. FEJEZET

Hibajavítási eszközök és technikák

Komplex programok készítése esetén számítanunk kell arra, hogy az utolsó programsor megírását követően még sok munka vár ránk. Akármennyire is ügyelünk a tervezésnél, akármennyire próbáljuk a részleteket összehangolni, általában a program nem lesz minden ízében működőképes. Ezzel az általános (és nem túl boldogító) igazsággal a programozó nap mint nap szembesül. Még a legmasszívabbaknak tűnő alkalmazásokat is valamilyen módon össze lehet zavarni. A valóságban arra kell törekedni (mivel tökéletes program nincs), hogy alkalmazásaink a lehető legstabilabbak legyenek – olyanra, amelyik még az összeroppanás előtt elmenti a felhasználó munkáját.

A kivételkezelők (l. előző fejezet) sokat segítenek a nem várt futási hibák elfogásánál, de természetesen nem pótolhatják a program megírása utáni funkcionális ellenőrzést és hibajavítást. Amennyiben az ellenőrző folyamat meggyőzött minket, hogy valahol a programba nem kívánt hiba csúszott, akkor minden elérhető eszközt mozgósítani kell a megtalálására. A Delphi fejlesztési környezete sok olyan segédeszközzel szolgál, amelyek segítségünkre lehetnek ebben a nem éppen könnyű feladatban. A fejezet célja ezek használatának bemutatása.

Természetesen a program futásának tesztelése, a hibakeresés és -javítás megkezdése előtt, ajánlatos egy biztonsági másolatot készíteni a forráskódról. Ezen kívül hasznos lehet formok ASCII reprezentációját (l. 4. fejezet) elkészíteni és eltárolni, így lehetőség nyílik a form és az ahhoz tartozó komponensek kezdeti tulajdonságait ellenőrizni.

Jogos elégedetlenséghez vezet, ha programunk lassabban fut a vártnál. Ebben a fejezetben olyan technikák is ismertetésre kerülnek, amelyek segítségével, a feladatához képest lassú programok futási sebessége nö-

velhető anélkül, hogy újabb hibákat ültetnénk beléjük (a kód „túloptimalizálása" az egyik leggyakoribb forrása a programozási hibáknak). A fejezetet a jó programozási stílus jellemzése zárja.

■ *Hibakereső eszközök és feladatkörük*

A Delphi mindkét változatában a Run menü tartalmazza a tesztelő és hibakereső eszközök nagy részét. Ezek közül sok az eszközpalettán is megtalálható. A hibakereső eszközök egy tipikus alkalmazását jelenti a program futásának időnkénti felfüggesztése. A 9-1. táblázat a Run menüből elérhető hibakereső funkciókat tartalmazza.

A View menün keresztül érhető el az a dialógusablak, amelyben a töréspontok (breakpoints) beállítását tehetjük meg, továbbá azok az ablakok, ahonnan a hasznos Call Stack és Watches funkciók (l. még a „A Watch funkció" részt) szabályozhatók.

Tipp: *A kódszerkesztő gyorsmenüje hozzáférést biztosít sok, a 9-1. táblázatban felsorolt eszközhöz. A gyorsmenüt a jobb oldali egérgombbal aktivizálhatjuk.*

■ *Hogyan készítsünk olyan programokat, amelyekben könnyű a hibákat megtalálni?*

Hosszú és összetett programok helyes működését nem könnyű ellenőrizni, de segíthet, ha a program írása közben szem előtt tartunk néhány egyszerű irányelvet. (Ezek a módszerek általában magát a programozást is egyszerűsítik.) Mindenekelőtt célszerű a programot olyan kezelhető részekre bontani, amelyek mindegyike egy külön feladatot valósít meg, az ilyen módon strukturált program lényegesen könnyebben tesztelhető és módosítható. Minden objektum, eljárás és függvény elkészítése után vizsgáljuk meg, hogy képesek-e fogadni és megfelelően feldolgozni minden lehetséges paramétert vagy paraméterkombinációt. A kívánt metódusokat és tulajdonságokat egybezáró (és újra felhasználható) azon objektumok (l. 7. fejezet), amelyeknek a hibamentességéről meggyőződtünk, sok munkát takaríthatnak meg számunkra.

9-1. táblázat. A Run menü hibakezelő funkciói

Eszköz	*Billentyűkombináció*	*Funkció*
Run	ALT-R-R (F9)	Elindítja/újraindítja a programot.
Parameters	ALT-R-P	Beállítja a futási paramétereket.
Step Over	ALT-R-S (F8)	Úgy működik, mint a Trace Into, csak a függvény- és eljáráshívások egy lépésben történnek.
Trace Into	ALT-R-T (F7)	A programot utasításonként futtatja.
Run to Cursor	ALT-R-C (F4)	Az aktuális kurzorpozíción található forrássorig futtatja a programot.
Show Execution Point	ALT-R-H	Megmutatja a sorra kerülő forrássort.
Program Pause	ALT-R-G	Felfüggeszti a program futását.
Program Reset	ALT-R-E (CTRL-F2)	Befejezi a lépésenkénti futtatást.
Add Watch	ALT-R-W (CTRL-F5)	Megjeleníti a Watch Properties dialógus-ablakot, ahol megathatjuk a követni kívánt változókat és kifejezéseket.
Add Breakpoint	ALT-R-B	Töréspontot helyez el, ami a futást elakasztja.
Evaluate/Modify	ALT-R-V (CTRL-F7)	Megjeleníti az Evaluate/Modify dialógus-ablakot, amelyben a programban szereplő kifejezések értékét nézhetjük meg és módosíthatjuk.

A következő lépés az ellenőrzött objektumok és eljárások összeillesztése, és együttes működésük tesztelése. Elképzelhető, hogy egyes eljárások és függvények működéséhez, még meg nem írt alprogramok eredményei szükségesek. Ekkor az ún. *csonk-programozás* (stub-programming) technikát alkalmazzuk, ahol a szükséges eredményeket konstansokkal helyettesítjük. A cél mindig az, hogy egy olyan programhoz jussunk, amely valamit csinál, még akkor is, ha csak későbbi működésének körvonalai vehetők ki. Definiáljuk a függvényeket és eljárásokat, de ahelyett, hogy ténylegesen csináltatnánk velük valamit, töltsük fel őket konstansokkal. Az eljáráshívások természetesen így is ugyanúgy működnek, de azok csak konstansokat fogadnak a csonkokból. A konstansok változtatásával teremthetünk újabb tesztkörülményeket.

Tegyük fel, hogy a programot a nyilvánvaló szintaktikai hibáktól mentessé és úgy-ahogy futtathatóvá tettük. Tesztelés során azonban kiderül, hogy a program nem úgy működik, ahogy kellene: tartalmaz javításra váró hibákat. Nem szabad ezen meglepődnünk vagy elkeseredünk: a hibák természetes velejárói a nagyobb méretű programoknak. Célunk ezek lokalizálása és a típusuktól függő gyógymód alkalmazása.

Két alapvető programhibát különíthetünk el: futási idejűt (run-time) és logikait. A futási időben bekövetkezett hibára példa egy nem létező állomány megnyitása.

Tipp: *Az első alkalommal, amikor egy progamban futási idejű hibákat keresünk, érdemes lehet a kivételkezeléskor bizonyos hasznos információkat kiíratni. Egy másik lehetőség a „Break on exception" funkció használata, amellyel megállapíthatóvá válik, hogy melyik sor idézte elő a kivételt.*

A logikai hibák egy hatalmas családot alkotnak, amelybe beletartozik minden olyan hiba, amely a program működési elvével ellenkezik. Ide sorolhatók többek között azok az eljárások, amelyek nem kommunikálnak megfelelően és a kód belső logikai hibái is.

Logikai hibák

A logikai hibák kijavításához először is lokalizálni kell őket – azaz meg kell találnunk a program azon részét, amelyik a problémákat okozza. Ha a moduláris elvnek megfelelően jártunk el, akkor ez a feladat jóval egyszerűbb. Az önálló egységként működő programrészek sokkal kisebb szénakazalt jelentenek a meglelendő tűnek. Hasonlóan azzal, hogy egy gyakori feladatot egy objektumon keresztül valósítunk meg, a hibakeresésnél is kevesebb dologgal kell törődnünk.

Amennyiben a fejlesztés egymást követő szakaszaiban rendszeresen teszteltük a programot, akkor már tisztában kell lennünk azzal, hogy melyik objektum, eljárás vagy függvény felelős a rendellenes működésért. Nyilánvaló, hogy egyszerűbb rámutatnunk a problémás objektumra vagy alprogramra, ha az alkalmazást magunk fejlesztettük, hiszen így ismerjük a program logikáját. Ha nem saját programról van szó, vagy vártunk az

alkalmazás „elkészültéig" a hibakereséssel, akkor az ismertetésre kerülő technikák alkalmazásával nézhetjük végig a modulokat.

Tételezzük fel, hogy egy hibás objektumot, eljárást vagy függvényt tesztelük. Három különböző eset léphet fel:

- A bemenő adatok rosszak – az objektumnak, eljárásnak vagy függvénynek átadott értékek összezavarják azt.
- A kimenő adatok rosszak – az objektum, eljárás vagy függvény hibás információkat szolgáltat a program többi részének (ami például nem tervezett mellékhatásokhoz vezethet).
- Valami az objektumon, eljáráson vagy függvényen belül nem működik (például túl sokszor hajt végre egy bizonyos műveletet, vagy nem a kívánt időben törli a képernyőt.)

Az első két esetben a hiba az alábbiak egyikére vezethető vissza: az eljárásnak vagy függvénynek (vagy az objektum metódusainak) átadott paraméterre, az eljárás vagy függvény belsejében a paramétereknek, tulajdonságoknak, vagy a unit (form)-szintű ill. globális változóknak történő értékadásokra.

Hogyan állapítjuk meg, hogy melyik szituációval van dolgunk? Először is, nehéz elképzelni egy helyesen megírt *kis* objektumról vagy alprogramról, hogy egy darab papír segítségével ne tudnánk követni annak menetét, és ne tudnánk megállapítani az elvégzett műveletek eredményét a legtöbb esetben. Egyszerűen „számítógépet kell játszanunk", a függvényekben és eljárásokban lévő műveleteket kézzel elvégezhetjük. (Ez azt is jelenti, hogy nem tételezhetünk föl olyan dolgokat, amiket a számítógép ne tudna a programrész azon pontján; nem ruházhatunk fel változókat bizonyos értékekkel, amíg a megfelelő értékadásoknak köszönhetően ténylegesen nem rendelkeznek velük.) Így ellenőrizhetjük, hogy a függvények, objektumok és eljárások azt csinálják-e, amit kell.

■ *Nyomkövetési információk*

Ahhoz, hogy használhassuk az integrált nyomkövetést, programjainkat úgy kell lefordítani, hogy a hibakereső össze tudja kapcsolni a keletkező .exe fájlt a forrásprogrammal. Ezt az ún. *szimbolikus nyomkövetési információk* hozzáadásával tehetjük meg. Ez nem más, mint egy olyan szimbólumtáblázat, amely összeköttetést teremt a program forráskódja és a

fordító által generált gépi kód között. A Delphi ezt alapértelmezés szerint beleépíti a kódba. A nyomkövetési információk hozzáadását a Project Options dialógusablak Compiler oldalán tehetjük lehetővé, illetve iktathatjuk ki.

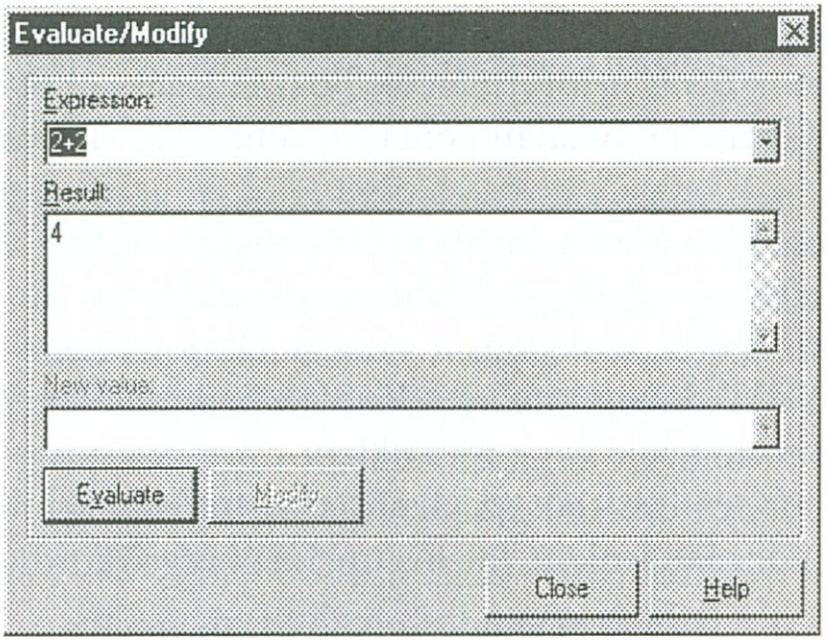
Ha a Turbo Debugger for Windows-t kívánjuk használni, akkor a végső .exe állománynak tartalmaznia kell szimbolikus nyomkövetési információkat a TDW számára. Ehhez szükséges a Project Options dialógusablak Linker oldalán a „TDW debug info" jelölőablak beikszelése. A szimbolikus nyomkövetési információk nagy mértékben növelik a program .exe állományának méretét, és lassítják az alkalmazás futását, ezért ne felejtsük el kikapcsolni a hozzáadásukért felelős opciót a végső fordítás előtt.

Megjegyzés: A Turbo Debugger for Windows egy különálló termék, amely nagyon sok speciális nyomkövetési (pl. hardver töréspontok) lehetőséggel egészíti ki a Delphi beépített hibakeresési funkcióit. Ebben a könyvben ezt nem tárgyaljuk. További információkat a beépített help, vagy a Turbo Debuggerhez mellékelt dokumentációk szolgáltatnak. Ez utóbbi kiegészítőt a Delphi RAD-packban, vagy a Delphi magasabb verzióihoz mellékelve találhatjuk meg. A Delphiben készített DLL-ek hibakeresését a Turbo Debugger-rel végezhetjük.

■ *Az Evaluate/Modify dialógusablak*

A legtöbb programnyelvben módunk van az utasítások, változók és kifejezések tesztelésére, ez alól a Delphi sem kivétel. Az Evaluate/Modify dialógusablakban pontosan ezt tehetjük meg. Az ilyen tesztelést általában töréspontok elhelyezése után vagy lépésenkénti futtatás alatt (Trace Into) használjuk. A 312. oldalon lévő ábrán az Evaluate/Modify dialógusablak látható.

A dialógusablakot a Run menün keresztül aktivizálhatjuk. Ez egy mozgatható modális komponens, azaz be kell zárnunk, mielőtt a fejlesztői környezet más eszközeihez, ablakaihoz hozzáférhetnénk. Ha a dialógusablakba begépeljük, hogy 2+2, majd lenyomjuk az ENTER-t, akkor a Result ablakocskában megjelenik az eredmény: 4. A szerkesztési sorba írt



szöveget a szokásos Windows-os módszerekkel vihetjük be és módosíthatjuk. Lehetőségünk van szövegrészt a Clipboard-ra másolni, és onnan beilleszteni a kívánt sorba. Az ENTER lenyomása után a Delphi megpróbálja kiértékelni a megadott kifejezést.

Az Expression lenyíló listáról bármely, előzőleg megadott sort kijelölhetünk, és az ENTER lenyomásával ismét kiértékeltethetünk.

Hogyan használjuk az Evaluate/Modify dialógusablakot nyomkövetésre?

Az Evaluate/Modify dialógusablak megkönnyíti az egyes változók és kifejezések tesztelését, segítségével összehasonlíthatjuk a várt és a program által kiszámolt eredményeket. A Result ablakban a kiértékelni kívánt változók ill. kifejezések aktuális értékét láthatjuk. A változót vagy kifejezést az Expression lenyíló listaablak szerkesztési sorába írjuk be. Ha a kódszerkesztőben kijelölünk egy változót, akkor az automatikusan megjelenik az Evaluate/Modify dialógusablakban.

Lehetőség van a változók értékét módosítani, ehhez először az Evaluate gombra klikkelünk, majd a kurzort a New value szerkesztési sorra (a Result ablak alatt) irányítjuk, ahol beírhatjuk a változó új értékét. (A New value ablak csak akkor aktivizálható, ha van minek megváltoztatni az értékét – l. ábra.) Természetesen nem árt, ha vigyázunk a módosításokkal, a nemkívánatos mellékhatások elkerülése érdekében.

Megjegyzés: Az Evaluate/Modify dialógusablakban végzett módosítások nincsenek hatással sem a forráskódra, sem a lefordított programra. (Ellenben a változók vagy tulajdonságok értékének megváltoztásával, természetesen a futó program is másként fog viselkedni.)

Ha az Evaluate/Modify dialógusablakban végzett vizsgálódás alapján úgy döntünk, hogy a forráskódot is módosítjuk, akkor ezt csakis kódszerkesztő megfelelő részében tehetjük meg.

■ *A Trace Into funkció*

Gyakran, ha papíron végigpörgettük a programot (vagy programrészletet), hasznos, ha ugyanezt a fejlesztői környezetben is megtesszük, ugyanúgy soronként, miközben a változók értékeinek az alakulását követjük. A Delphiben ezt a léptetést a Trace Into funkció valósítja meg. Az F7 vagy az eszközpaletta Trace Into gombjának ismételt lenyomásával hajthatjuk végre az egymást követő utasításokat – egy leütés, egy utasítás. (Természetesen, ha a Delphi egy esemény bekövetkezésére vár, akkor a léptetéskor sem történik semmi.) Ha a végrehajtás egy függvényhívásra kerül, és a hívott függvényt nyomkövetési információk hozzáadása nélkül fordítottuk, akkor a Trace Into az alprogramot egy lépésben hajtja végre, majd a hívás utáni sorra ugrik.

A léptetés elindításakor, a program első végrehajtható utasítása (általában a form OnCreate eseménykezelője) kiemelve jelenik meg a kódszerkesztő ablakban. Az F7 minden további lenyomásával végrehajtódik a kiemelt utasítás, majd a kiemelés a következő utasításra ugrik. A léptetés alkalmas a program logikai nyomon követésére, a döntési szerkezetek és eljárások ellenőrzésére.

Amikor a léptetés egy eljáráshíváshoz ér, kódja megjelenik a kódszerkesztő ablakban. Utasításainak egyesével történő végrehajtása után, a hívó rutin lesz újra látható a kódszerkesztőben.

A léptetést nem csak a Trace Into funkcióval (F7 vagy az eszközpaletta megfelelő gombja), hanem a Step Over (F8) funkcióval is végezhetjük. Ez az eszköz minden eljárást és függvényt egy utasításnak tekint. Azokban az esetekben, amikor bizonyos eljárások és függvények működőképességéről meggyőződtünk, előnyösebb lehet a Step Over használata.

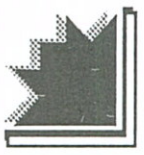
■ *A program futásának felfüggesztése*

Gyakran kényszerülünk a program ideiglenes megállítására azért, hogy a program állapotáról, azaz a változók értékéről képet kaphassunk. Tegyük fel, hogy meg szeretnénk tudni, hogy az egyik változó miért nem a megjósolt értékkel rendelkezik. Ehhez először valamilyen módon meg kell találnunk, hogy a programban hol kezd el a változó „furcsán" viselkedni. Az értékek kiiratása az Evaluate/Modify dialógusablakban most nem biztos hogy segít.

Kétféleképpen érhetjük el, hogy egy program ideiglenesen megálljon: a Run menü Program Pause pontjával, vagy töréspontok (breakpoints) használatával. A töréspontokat a forráskód azon soraihoz rendeljük, amelyekben a futást fel kívánjuk függeszteni. A töréspontokat be- és kikapcsolhatjuk az F5 lenyomásával vagy a kódszerkesztő gyorsmenüjén keresztül. (Használhatjuk még a Run menü Add Breakpoint pontját is.) A töréspontok pirosan jelennek meg a kódban. Az Edit Breakpoint dialógusablakban egy sor beállítás közül választhatunk a töréspontok létrehozásakor. A Pass Count szerkesztési ablakba beírt értékkel azt szabályozhatjuk, hogy hányadik áthaladáskor álljon le a Delphi az adott sornál. Az alkalmazás futása során, ha a Delphi egy töréspontot talál, akkor még az ahhoz tartozó forrássor végrehajtása előtt felfüggeszti a futást, és a program ún. *stop módba* kerül.

Több töréspontot is elhelyezhetünk egyszerre. Egy adott töréspontot úgy törölhetünk, hogy annak sorába helyezzük a kurzort és lenyomjuk az F5-öt. Amennyiben az összes töréspontot le szeretnénk törölni, akkor válasszuk a View menü Breakpoints option pontját, majd a Breakpoint List ablakban nyomjuk le a jobb oldali egérgombot, és válasszuk a Disable All Breakpoints opciót.

Megjegyzés: *Ha a programot csak egy helyen szeretnénk megállítani, akkor használhatjuk a Run | Run to Cursor pontot (F4).*

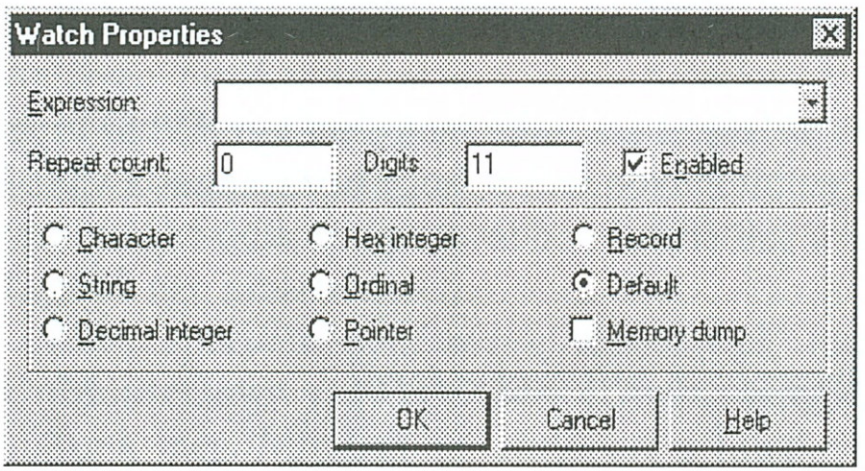


Tipp: *A Run | Run ponttal folytathatjuk egy leállított program futását. Az F8 vagy az eszközpaletta Step Over funkciójának választásával a függvényeket és eljárásokat a Delphi egy utasításként kezeli. Ilyen módon nem kell a függvények illetve eljárások összes során végig lépdesni, ha tudjuk, hogy ott nincs probléma. Ha emellett a View*

menü Call Stack funkcióját is aktivizáljuk, akkor megtudhatjuk, hogy melyik eljárás hívta azt az eljárást, amelyikben éppen vagyunk, sőt mi több, nyomon követhetjük az egész eljáráshívási láncot is, ha szükséges.

■ *A Watch funkció*

A hibakeresés lényege – bármilyen programról is legyen szó – annak az objektumnak, függvénynek vagy eljárásnak a megtalálása, amelyik nem működik. Ha ez megvan, akkor az adott alprogramban vagy objektumban keresendő tovább a hiba. Habár az Evaluate/Modify dialógusablak léptetés során lehetőséget nyújt kifejezések kiértékelésére, a Delphi Watch funkciója sokkal hatékonyabb eszköz. A program futása előtt, vagy a futás ideiglenes felfüggesztésekor megadhatjuk, hogy a Delphi melyik változót, kifejezést vagy feltételt kövesse végig. Ennek egyik módja a Watch Properties dialógusabak használata, amelyet a Run menü Add Watch pontjával jeleníthetünk meg. A dialógusablakot az alábbi ábra mutatja:



Ahogy a képen látható, felbukkanáskor a Watch Properties dialógusablak Expression szerkesztési sora üres. Ide írjuk a követni kívánt kifejezést. Az egyes opciókkal szabályozhatjuk, hogy a Delphi milyen típusúnak jelenítse meg a kifejezést.

Megjegyzés: *Kizárólag az aktuális formhoz vagy unithoz tartozó globális változókat és az azokból felépített kifejezéseket követhetjük. Másfajta kifejezések használatakor „process not accessible" üzenetet kapunk.*

A nyomon követett változókat és kifejezéseket a Delphi a Watch List ablakban jeleníti meg. Az itt látható értékek mindig a program állapotának megfelelően alakulnak. Ha egy adott változót már nem kívánunk nyomon követni, akkor azt a CTRL-D billentyűkombinációval, vagy a gyorsmenün keresztül törölhetjük az ablakból.

Gyors értéklekérdezés

A kódszerkesztő gyorsmenüjének Add Watch at Cursor pontjának (CTRL-F5) választásával, megnézhetjük a kurzor aktuális pozícióján lévő változó vagy kifejezés értékét. Ez a gyors értéklekérdezésnek (instant watch) hívott funkció a Delphi Evaluate/Modify dialógusablakát egészíti ki, amelyben azon eljárás változóinak az értékét kérdezhettük le, amelyben a programvégrehajtást felfüggesztettük.

A gyors lekérdezés használatának a módja:

1. Helyezzük a kurzort a lekérdezni kívánt változóra vagy kifejezésre, majd jelöljük ki azt az egér húzásával, vagy a megfelelő SHIFT-NYÍL kombináció használatával.
2. Válasszuk a gyorsmenü Add Watch at Cursor pontját (ALT-F10-A vagy CTRL-F5).

A Delphi jelzi, ha az adott értéket nem tudja megjeleníteni. Ekkor vagy bezárjuk az ablakot az ESC gomb lenyomásával, vagy hozzáadjuk a változót a Watch List ablakhoz.

■ *Néhány végső megjegyzés a hibakeresésről*

A függvények és eljárások paramétereinek változtatgatása, és a többi itt leírt hibakeresési technika használata nem tekinthető csodaszernek. Semmilyen trükk nem segíthet, ameddig nincs elképzelésünk arról, hogy mit is kellene az adott alprogramnak csinálnia. Lehet hogy rossz feltételt használunk az if-then utasításban? Lehet, hogy >= helyett = -t kellene írnunk? Használjuk a Watch funkciót azon logikai relációk értékének (True vagy False) tesztelésére, amelyek, úgy tűnik, soha nem teljesülnek (teljesen helyénvaló a Watch Property ablak Expression sorába olyat írni, mint x=19). Vizsgáljuk meg a ciklusokat, amelyek gyakori hibaforrásokat

jelentenek. A számlálók helyesen vannak inicializálva (lehet, hogy eggyel elszámoltuk magunkat)? Nem lehet, hogy elöltesztelő ciklus helyett hátultesztelőt kellene használni vagy fordítva?

Tipp: *Ha egy Repeat- vagy While-ciklus túl sokáig fut, akkor érdemes egy ideiglenes számlálót beiktatnunk. A Watch funkció segítségével állítsuk le a ciklust, amikor a számláló túl nagy értéket vesz fel. Ekkor vizsgáljuk meg a kilépési feltételt és annak alkifejezéseit, és próbáljuk megállapítani, hogy az miért nem teljesül.*

■ *Eseményvezérelt hibák és problémák*

Eseményvezérelt programok hibakeresésénél fontos, hogy tisztában legyünk az olyan lehetséges problématípusokkal, amelyek régebbi programozási nyelvekben nem fordulhattak elő. Ezek közül a leggyakoribb az ún. *eseményzuhatag* (event cascade). Ez a fajta hiba abból fakad, hogy egyes eseménykezelő eljárások saját magukat, vagy másik eseménykezelőket hívnak meg vég nélkül, anélkül, hogy ki lehetne kerülni belőle valahogyan. Az ilyen hibákat leginkább olyan helyeken kereshetjük, ahol a vezérlőelem OnChange eseményeljárását átírtuk. Az eseményeljárás meghívódik, belül valamit változtatunk a komponensen, ami azt eredményezi, hogy az újra meghívódik és így tovább – elméletileg a végtelenségig, gyakorlatilag kapunk egy hibaüzenetet.

Más, speciális nehézségek lépnek fel, amikor egy programot egy OnMouseDown vagy egy OnKeyDown eseménykezelő végrehajtása során állítunk meg. Mindkét szituációban, a hibakeresés alatt az ember természetesen felengedi azt az egérgombot ill. billentyűt, amely kiváltotta az eseménykezelő meghívását. Amikor azonban a programot folytatjuk, akkor a Delphi feltételezi, hogy az egérgomb vagy billentyű még le van nyomva, így a fontos OnMouseKeyUp illetve az OnKeyUp eljárások nem lesznek soha sem meghívva.

■ *Programozási stílus*

Egy darabig emlékezünk komplikáltabb programjaink felépítésére, örökre azonban nem marad a fejünkben. A jó dokumentáció éppen ezért kulcsfontosságú. Néhányan kihasználják a megjegyzések adta lehetőségeket, és végigkommentálják a programot. Jelentéssel bíró változó nevek használatával kiegészítve, valóban ez jelenti a legjobb dokumentálási módot. Rögzítsünk bizonyos konvenciókat a globális, formszintű és lokális változók neveinek írásmódjáról, és ne térjünk el azoktól. Kerüljük el a trükkös kódokat, ha mégis különlegesen okos módon oldunk meg valamit, akkor fontos a megfelelő kommentálás. (Legtöbb esetben azt találjuk, hogy az okos kódrészletre nincs is igazán szükségünk.) Semmi nem nehezebb, mint hat hónap elteltével megváltoztatni egy „rafinált" kódot.

Amikor arra koncentrálunk, hogyan lehetne valamilyen trükköset elkövetni, hogy felgyorsítsuk a programot, könnyen megfeledkezünk arról az elsődleges programozási szempontról, hogy a programoknak stabilaknak kell lenniük. Valójában a nagy sebességugrások soha sem a kis „csavaroknak" köszönhetőek, hanem a helyes algoritmusválasztásnak. Az algoritmusok tulajdonképpen problémák megoldására használt módszerek.

Ez nem azt jelenti, hogy biztonságosan működő programokon nem lehet változtatni, hogy azok gyorsabban fussanak. Például egész változók használatának bevezetése, ahol lehetséges, nem jelent veszélyt a program stabilitására. Ne használjuk a memóriaemésztő variant típusú változókat, hacsak nincs szükségünk speciális tulajdonságaikra.

Próbáljuk ki az alábbi technikákat a program sebességének megnövelésére, ha szükséges (természetesen a RAM méret és a Winchester sebesség sem utolsó szempont):

- Olvassuk be előre a DLL-eket, ha lehetséges.
- Szüntessük meg a formokat, ha már nincs rájuk szükség.
- Töröljük ki azt a kódot, amit már nem használunk a projektben. Többek között a végső fordítást a szimbolikus nyomkövető információk hozzáadása nélkül végezzük.
- Használjunk lokális változókat, amikor csak lehet.
- Lehetőleg használjunk konstans paramétereket.

Bárhogyan is, egy olyan program (még ha azt mi írtuk is valamikor) megváltoztatása vagy hibajavítása rendkívül nehéz, amelyet egyáltalán nem vagy csak alig kísér megjegyzés, dokumentáció, és amelyben infor-

mációtartalommal nem rendelkező változó nevek vannak. Egy CsináljMartinit (Jéggel,CitromNélkül) nevű eljárás egy James Bondról szóló programban lehet, hogy elmegy (talán még ott sem), de nem lenne túl beszédes egy trigonometriai függvényekkel dolgozó alkalmazásban. Mivel a Delphi megengedi a hosszú változónevek használatát, ezért csak a saját dolgunkat nehezítenénk olyan változónevekkel, mint X, X13, X17, X39 stb. Ha átláthatóságra törekszünk a programjainkban ahelyett, hogy elsősorban a hatékonyság miatt aggódnánk, akkor sokkal jobban járunk.

Legvégül, ha egy függvény, eljárás vagy objektum jól működik, akkor azt mentsük el, hogy később is fel tudjuk őket használni. A Pascal unitjainak egyik nagy előnye, hogy segítségükkel a vett, vagy általunk készített eszköztárakat akár minden programunk írásánál felhasználhatjuk. Komplikáltabb programok általában sok olyan függvényt és eljárást tartalmazhatnak, amelyek már korábban is előfordultak esetleg más környezetben. Ez azt jelenti, hogy a felület kialakítása után gyakran nem kell mást csinálnunk, mint az előre megírt, hibamentes függvény- és eljáráskönyvtárak részeit módosítani, és beilleszteni a felület eseménykezelőibe. (Ez az egyik oka annak, hogy a kereskedelmi forgalomban lévő eszköztárak a Delphihez miért olyan népszerűek.)

10. FEJEZET

Fájlkezelés

Ez a fejezet azt mutatja be, hogyan kezelhetjük a fájlokat és a háttértárakat a Delphi alkalmazásokban. Elsőként azokkal a parancsokkal foglalkozunk, amelyeket a DOS dolgoz fel – például átnevezhetünk fájlokat, megváltoztathatjuk az aktív meghajtót, és mozoghatunk a könyvtárak között a Delphi közvetítésével. Láthatunk olyan Delphi parancsokat, amelyek megkönnyítik a dolgunkat, amikor állományokkal dolgozunk – ilyen például a szabad diszkkapacitás megállapítása. Ezt követően megtanulhatjuk a fájlrendszer komponensek használatát, amelyek mindkét Delphi változatban megtalálhatók a komponenspaletta System oldalán. Ezután ízelítőt kaphatunk a Delphibeli fájlkezelésről. A Delphi ASCII fájlok és bináris állományok kezelését egyaránt lehetővé teszi, a fejezetben mindkét típussal kapcsolatos technikák bemutatásra kerülnek. (Annak ellenére, hogy a Delphi sok adatbáziskezelési eszközzel rendelkezik – amelyeket majd a 14. fejezetben tárgyalunk –, az adatbázisok használatánál szükségünk lesz fájlok léterhozására és közvetlen manipulálására.) Az állománykezelés mindkét Delphi verzióban lényegében ugyanúgy működik. Az igazi kivételt természetesen az jelenti, hogy a Delphi 32 támogatja a Windows 95-ben és a Windows NT-ben használható hosszú fájlneveket.

Megjegyzés: *Amikor fájlokkal dolgozunk, gyakran érdemes a Delphi könyvtári dialógusablakait használni. Ezekről a 13. fejezetben lesz szó. Fontos szem előtt tartanunk, hogy a Windows felhasználók elvárják a szokásos dialógusablakok megjelenését, amikor különböző fájlműveleteket végeznek.*

■ *Könyvtár- és állománykezeléssel kapcsolatos parancsok*

A Delphiben öt olyan eljárás és két olyan függvény áll rendelkezésünkre, amelyek közvetlenül az operációs rendszer rutinjait használják, ezek nevei megegyeznek a megszokott operációs rendszerbeli parancsokéval.

Megjegyzés: Az olyan eljárás vagy függvény, amely az operációs rendszert használja valamilyen fájlkezelési feladat megoldására, kivételt vált ki, ha az operációs rendszer nem tudja végrehajtani a rábízott műveletet. A 8. fejezetben láthattuk, hogyan kell bánni az ilyen jellegű futási hibákkal.

Az első három ilyen parancs paramétereként egy sztringet, vagy sztringkifejezést kell megadnunk. Például az

```
MkDir('TESTDIR');
```

parancs egy TESTDIR nevű alkönyvtárat adna az aktuális könyvtárhoz. A

```
MkDir('C:\TESTDIR');
```

sor pedig egy alkönyvtárat helyezne el a C meghajtó könyvtárrendszerének gyökerében. A

```
MkDir ('C:\Gary Program Files\');
```

egy hosszú könyvtárnevet hozna létre a Delphi 32-ben.

A Rename és az Erase parancs használatához először egy speciális *File* típusú változót kell létrehoznunk, amellyel majd a megnyitott fizikai állományra hivatkozhatunk. Az alábbi példa, amely egy GOODFILE-t átnevez OLDFILE-lá, ezt mutatja be:

```
var
    FileName : File;
begin
    AssignFile (FileName, 'GOODFILE');
```

```
    Rename (FileName, 'OLDFILE');
    CloseFile (FileName);
end;
```

Megjegyzés: *Ez a példa jól demonstrálja az AssignFile eljárást. Gyakorlatban előnyösebb lehet a RenameFile függvény használata, amely az átnevezés sikerességétől függő logikai értéket ad vissza.*

10-1. táblázat. A Delphi DOS műveleteket használó parancsai

Parancs	*Funkció*
ChDir	Az aktuális könyvtárat és a lemezmeghajtót megváltoztatja.
MkDir	Egy új könyvtárat hoz létre.
RmDir	Megszünteti a megadott könyvtárat.
Rename	Egy fájl nevét változtatja meg. (A RenameFile függvény ugyanezt csinálja, visszatérési értéke pedig a művelet sikerességére vagy sikertelenségére utal – True/False.)
Erase	Egy állományt töröl a lemezről. (A DeleteFile függvény ugyanezt csinálja, visszatérési értéke True, ha a törlés sikeres volt.)

Az AssignFile eljárás egy külső állománynevet kapcsol egy előzetesen deklarált File típusú változóhoz. A kapcsolat mindaddig megmarad, ameddig be nem zárjuk az állományt.

Pascal tipp: *Használható a régebbi Assign eljárás is, a Delphi fejlesztői azonban az AssignFile eljárás alkalmazását ajánlják, mert az Assign összeütközésbe kerülhet az ugyanilyen nevű metódussal.*

A fájlváltozó értéke egy valóságos fájlnév kell hogy legyen az adott operációs rendszerben. A Delphi 16-ban a fájlnevek használatára a DOS-ban érvényes szabályok vonatkoznak. A Windows 95 és a Windows NT a Delphi 32-beli fájlneveknek nagyobb szabadságot biztosít.

- A Delphi 16-ban: nyolc karakter plusz három karakter a kiterjesztésnek (ponttal elválasztva).
- A Delphi 32-ben: 255 karakter és a szóköz is megengedett.
- Mindegyik operációs rendszerben az alábbi karakterek használhatók:

A–Z, 0–9 ( ) { } @ # $ % & ! - _ ' / ~

Megjegyzés: A fájlkezelő parancsok gyakran elfogadják a szokásos DOS helyettesítő jeleket. Például az Erase ('.*') minden fájlt töröl az aktuális könyvtárból (ne bánjunk az ilyenekkel hanyagul).*

Más fájlkezelő és I/O függvények

Egyes fájlkezeléssel kapcsolatos feladatok olyan gyakoriak, hogy a Delphi fejlesztői jobbnak látták beépíteni a nyelvbe, kikerülve ezzel a Windows API és DOS rutinok hivásának szükségességét. (Azok számára, akik ismerik a DOS-t: ezek zöme az Int21H-t használó függvényeket helyettesíti.)

Például, a DiskFree függvény segítségével lekérdezhetjük, hogy mennyi szabad terület van még a megadott meghajtón. Delphi 16-beli szintaxis így néz ki:

function DiskFree (*Drive*: Byte):LongInt {Delphi 32-ben a visszatérési érték Integer típusú}

A 0 az aktuális meghajtónak, az 1 az A-nak, 2 a B-nek, 3 a C-nek stb. felel meg. A függvény visszatérési értéke a szabad diszkterületet adja meg byte-ban.

A GetDir eljárás egy sztringgel tér vissza, amely az aktuális könyvtár nevét tartalmazza a megadott meghajtón. A Drive nevű sztring létrehozása és a

```
GetDir(2, Drive);
```

utasítás végrehajtása után a Drive sztring paraméter értéke az aktuális könyvtár neve lesz a C meghajtón. Ennek az eljárásnk az általános szintaxisa így néz ki:

procedure GetDir(*D*: Byte; var *S*:String)

Ha a meghajtó nem létezik, akkor az eljárás az S parméter sztringnek az 'X:\' értéket adja.

Most ezen Delphi függvények legfontosabbjait mutatjuk be rövid leírást adva funkciójukról. (A beépített help-ben a többi függvényről is felvilágosítást kaphatunk.)

ChangeFileExt. Ez a függvény megváltoztatja a megadott állomány kiterjesztését (hasznos például biztonsági másolatok készítésénél). Szintaxisa így néz ki:

```
function ChangeFileExt (const FileName, Extension: string): string;
```

Az *Extension* paraméter tartalma lesz az új kiterjesztése a *FileName* paraméterben szereplő fájlnak.

DiskSize. Ez a függvény egy LongInt (a Delphi 32-ben Int) típusú változót ad vissza, amely a megadott meghajtó méretét tartalmazza. Szintaxisa a Delphi 16-ban így néz ki:

```
function DiskSize (Drive: Byte): Longint {a Delphi 32-ben a
visszatérési érték Integer típusú}
```

ahol, mint mindig, az aktuális meghajtónak a *Drive* paraméterben a 0 érték felel meg, az A meghajtónak az 1 és így tovább.

ExpandFileName. Ez a hasznos függvény egy sztringet ad vissza, amely a megadott fájl teljes elérési útvonalát tartalmazza. Használata:

```
function ExpandFileName (const FileName: string): string;
```

ExtractFileExt, ExtractFileName. Ezek a függvények paraméterként egy fájlnevet várnak, és visszatérési értékként egy kiterjesztést, illetve egy kiterjesztés nélküli fájlnevet adnak meg.

ExtractFilePath. Gyakran szükségünk van egy megadott fájl elérési útjára. Ennek a függvénynek a paraméterében egy fájlnevet adunk meg, vissszatérési értékként egy diszkazonosítót és mindazon könyvtárak nevét kapjuk meg, amelyek a fájlhoz vezetnek.

FileAge, FileGetDate, FileDateToDateTime. A FileAge függvény a fájl korát adja vissza; a FileGetDate megmondja keletkezési idejét. Mindkét függvény visszatérési értéke Longint típusú. A FileDateToDateTime függvény használatával, ezt a Longint típusba megadott időt TDateTime típu-

súvá alakíthatjuk (aminek következtében a Delphi dátum- és időkezelő rutinjai fogadni tudják).

FileExists. Ez egy rendkívül hasznos logikai függvény, amely True (igaz) értéket ad vissza, ha a fájl létezik, és False-ot (hamis), ha nem. Használata:

function FileExists(const *FileName*: string): Boolean;

A FileGetAttr függvény. A FileGetAttr függvény visszatérési értéke Integer típusú. Bitmaszkolási technikákkal és a „fa" konstansok használatával (faReadOnly, faHidden stb.) megtudhatjuk az adott fájl beállított attribútumait. Használata:

function FileGetAttr (const *FileName*: string): Integer;

FileSearch. Ezzel a függvénnyel egy fájlt kereshetünk a megadott könyvtárakban. Szintaxisa:

function FileSearch(const *Name*, *DirList*: string): string;

ahol a *Name* paraméterben a keresendő fájl nevét, a *DirList* paraméterben a könyvtárak listáját (pontosvessző az elválasztó karakter) adjuk meg. A visszaadott sztring vagy üres, vagy a teljes elérési útvonalat tartalmazza. Példa:

```
if FileSearch('CH1', 'C:\; D:\') = '' then
  ShowMessage ('A fájl elveszett!');
```

A FileSetAttr függvény. A FileSetAttr függvény segítségével a megadott fájl attribútumait állíthatjuk be. A lehetséges attribútumokat ugyanazokkal a „fa" konstansokkal állíthatjuk be, amelyeket a FileGetAttr függvénynél is használtunk. Szintaxisa:

function FileSetAttr(*FileName*: string, *AttributeValue*: Byte);

ahol az *AttributeValue* paraméterében adjuk meg a *FileName* nevű fájl attribútumait. Példa:

```
FileSetAttr (FileName, faReadOnly + faHidden):
```

Ez a sor elrejti az állományt és írásvédetté is teszi.

Tipp: A FilesetAttr függvénnyel rejtjük el azokat a fájlokat, amelyekről nem kell a felhasználónak tudni.

Például egy titkosított jelszó rejtett fájlban történő tárolása, amelyet a jelszó bekérésekor megvizsgálunk, gyakori (és viszonylag biztos) módszer arra, hogy az adott programhoz csak a jogosultak férjenek hozzá.

FindFirst, FindNext. Ezzel a két függvénnyel megadott attribútumokkal és névvel rendelkező fájlt kereshetünk a megadott könyvtárban. A FindFirst szintaxisa:

function FindFirst(const *Path*: string; *Attr*: Word; var *SearchRec*: TSearchRec): Integer;

A Path paraméterben az elérési útvonalat adjuk meg (tartalmazhat helyettesítő jeleket is). Az Attr paraméterbe a megfelelő (és korábban már látott) „fa" konstansok kombinációját írjuk. A *SearchRec* egy TSearchRec rekord típusú paraméter, amely a keresés eredményét tárolja. A Delphi 16-ban ez a típus a következőképpen van definiálva:

```
TSearchRec = record
  Fill: array[1..21] of Byte;
  Attr: Byte;
  Time: LongInt;
  Size: LongInt;
  Name: string[12];
end;
```

A Delphi 32-ben ugyanez így néz ki:

```
TSearchRec = record
  Time: Integer;
  Size: Integer;
  Attr: Integer;
  Name: TFileName;    {Ez egy sztringtípus}
  ExcludeAttr: Integer;
  FindHandle: THandle;
  FindData: TWin32FindData;
end;
```

A visszatérési érték kiértékelésekor ezen rekord mezőit kérdezzük le; ha a fájl nevére vagyunk kíváncsiak a Name mezőt, ha a méretre, akkor a Size mezőt. A FindNext szintaxisa:

function FindNext(var *F*: TSearchRec): Integer;

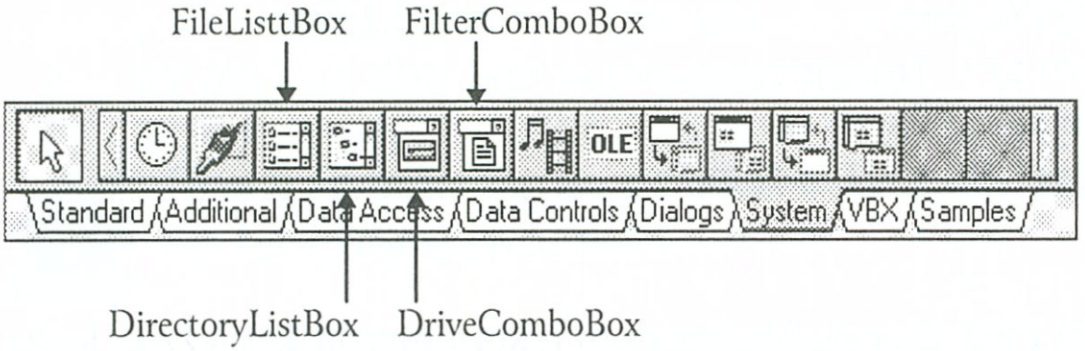
A FindNext függvény az előző FindFirst függvényben megadott paraméterek alapján végzi el a következő passzoló fájl keresését.

■ *Fájlrendszer-komponensek*

A fájlrendszer-komponensek a Delphi mindkét változatában hasonló funkciókat valósítanak meg: a felhasználó kiválaszthatja az aktuális lemezmeghajtót, megnézheti egy adott könyvtárhoz tartozó állományokat, megvizsgálhatja egy tetszőleges lemez könyvtárhierarchiáját. Mint minden Delphi komponenshez, ha komolyan ki akarjuk aknázni a bennük rejlő lehetőségeket, ezekhez is kódot kell írnunk. Ha el szeretnénk érni, hogy a felhasználó beavatkozásának (pl. bal oldali egérgomb lenyomása) eredményeképpen, a DOS könyvtárat vagy meghajtót váltson, akkor olyan eseménykezelő eljárásokat kell írnunk, amelyek a fejezet első részében ismertetett parancsokat használják.

Megjegyzés: A fájlrendszer-komponensek teszik teljessé a 13. fejezetben bemutatott gyakori dialógusablakokat.

Az ábrán a Delphi 32 komponenspaletta System (rendszer) oldala látható, amely fájlrendszer komponenseket is tartalmazza. (A 16 bites változatban is ugyanezeket az ikonokat találjuk.)



A fájlrendszer-komponenseket a legtöbbször együtt használjuk. A fejlesztők is ennek megfelelően alakították ki működésüket. Az ezeket kezelő programrésszel ellenőrizzük például, hogy mit csinált a felhasználó a DriveComboBox-szal, majd a szerzett információt továbbítjuk a DirectoryListBox-hoz. A DirectoryListBox-ban történt változásokat végül a FileListBox komponenshez juttatjuk.

■ *A FileListBox komponens*

A FileListBox listaablak jeleníti meg az aktuális könyvtárban található állományokat. Mint minden listaablakban, itt is lehetőségünk van a vezérlőelem helyének, méretének, színének és fontjellemzőinek a beállítására, fejlesztési időben, és a programon keresztül egyaránt. A FileBox legtöbb tulajdonsága megegyezik a közönséges listaablakokéval. Ha például az ablakban nem férnek el a listaelemek, akkor automatikusan megjelenik a függőleges görgetősáv, ami által a felhasználó elérheti a listához tartozó, de éppen nem látható elemeket is.

A FileListBox komponens minden olyan eseményre kész válaszolni, amelyet a listaablakok észlelni képesek. Emellett írhatunk olyan eseménykezelőket, amelyek egy gombnyomásra, vagy egy meghatározott egérmozgásra aktivizálódnak. Fontos ragaszkodnunk ahhoz a Windows konvencióhoz, hogy a fájlokat kettő és nem egy kattintással választjuk ki a listáról. Ellenkező esetben a nyilak minden lenyomása, amelyekkel egyébként a listán való lépegetést végezhetnénk, meghívná az egérhez

kapcsolt OnClick eseménykezelőt. (A listaablakoknál, a nyilakkal történő mozgás funkcionálisan ekvivalens az egérrel való kattintással.)

Tipp: Gyakran a FileListBox Items és ItemIndex tulajdonságát használjuk a GetDir parancs helyett.

A FileListBox komponens speciális tulajdonságai

A FileListBox FileType tulajdonságát használjuk arra, hogy csak bizonyos attribútumokkal rendelkező fájlokat jelenítsünk meg: például csak az írásvédetteket, vagy olyanokat, amelyeknek Archive bitje be (vagy ki) van kapcsolva (ez a bit jelzi, hogy az utolsó módosítás alkalmával készült-e biztonsági másolat a fájlról). A FileType tulajdonság valójában hét logikai tulajdonságból áll, ezek az ftArchive, ftHidden, ftNormal, ftReadOnly, ftSystem, ftVolumeID és az ftDirectory. Ez a tulajdonsághalmaz szabályozza, hogy milyen típusú fájlok jelenjenek meg a FileListBox-ban. Alapértelmezésben az ftNormal értéke True a többi haté False.

A FileListBox legfontosabb tulajdonságai közé tartozik még a Mask, a FileName és a Directory. A Mask tulajdonság határozza meg, hogy mely fájlok jelenhetnek meg a FileListBox-ban. A tulajdonság értéke egy sztring, amely tartalmazhat standard DOS helyettesítő jeleket is, úgymint a *-ot, amely bármilyen érvényes karaktersorozatot helyettesít és a ?-et, amely egy tetszőleges érvényes karaktert helyettesít. Alapértelmezésben a Mask értéke *.*, azaz nem válogat a fájlok között. (A Mask tulajdonságnak természetesen a FileType-nak megfelelő fájlokra van hatása.)

A Directory és a FileName tulajdonsággal lekérdezhetjük és beállíthatjuk a FileListBox aktuális útvonalát, az aktuális DOS könyvtárat ez nem befolyásolja. Ha azt meg szeretnénk változtatni a Delphiből, akkor használjuk a ChDir parancsot. Gyakran azonban nem akarjuk megváltoztatni az aktuális útvonalat ahhoz, hogy egy másik könyvtár tartalmát megvizsgáljuk. A Directory és a FileName tulajdonságok értékének megváltoztatásakor a Delphi aktivizálja a FileListBox OnChange eseménykezelőjét, ha van ilyen.

■ *A DirectoryListBox komponens*

A DirectoryListBox komponens az aktuális meghajtó könyvtár-rendszerét jeleníti meg. Az aktuális könyvtár mellett egy nyitott mappa található. Az alkönyvtárak mellett egy-egy összecsukott mappa áll, és az aktuális könyvtár feletti könyvtárakat nyitott, de nem árnyékolt mappa kíséri.

Megjegyzés: Amikor a felhasználó rákattint egy könyvtárra az egérrel, akkor az kiemelve jelenik meg, a kiemelés a listaelemek közti lépdeséskor automatikusan vándorol. Egy könyvtárra történő kettős kattintás után az átveszi az aktuális könyvtár szerepét, és a Delphi ennek megfelelően felfrissíti a listaablakot.

■ *A DriveComboBox komponens*

Ellentétben a FileListBox és a DirectoryListBox komponensekkel, a DriveComboBox vezérlőelemek lenyíló ablakok. Az ablakban az aktuális meghajtó jele látható. Amikor a felhasználó a nyílra kattint, akkor a Delphi lenyitja a listát, amelyben az összes háttértár fel van sorolva.

A DriveComboBox legfontosabb tulajdonsága a Drive, amellyel az aktuális meghajtót állíthatjuk be, illetve kérdezhetjük le. A DriveComboBox és a DirectoryListBox összehangolása úgy történhet, hogy a meghajtó- és a könyvtárkomponenseket összekötjük a DirList tulajdonságon keresztül. Ez a következőképpen néz ki:

```
DriveComboBox1.DirList := DirectoryListBox1;
```

A FilterComboBox komponens

A FilterComboBox komponenssel szabályozhatja a felhasználó a FileListBoxban megjelenített fájlok típusát. Miután a FilterComboBox FileList tulajdonságának értékül adtuk a kívánt FileListBox nevét, állítsuk

be a Filter tulajdonságon keresztül a megjeleníteni kívánt fájltípusokat. Például:

```
FilterComboBox1.Filter := 'Dokumentumok (*.doc) |
*.doc | Szövegfájlok (*.txt) | *.txt';
```

A fájlkomponensek összekötése

Amikor az összes fájlrendszerkomponenst egy formon használjuk, akkor biztosítanunk kell ezen komponensek egymás közti kommunikációját. Ehhez nem kell mást csinálnunk, mint a DriveComboList.DirList és a DirectoryListBox.FileList tulajdonságokat beállítani.

Az egyes fájlrendszerkomponensek közötti kapcsolatot program segítségével is létrehozhatjuk. Például, ha a felhasználó egy új meghajtót választ, akkor a Delphi aktivizálja a DriveCombBox OnChange eseménykezelőjét. Ezt kihasználva a következőket tehetjük:

1. A DriveComboBox OnChange eseménykezelőjében, a DirectoryListBox.Drive tulajdonságába másoljuk be a DriveComboBox.Drive tulajdonságának értékét. Ez megváltoztatja a DirectoryListBox-ban megjelenteket, ami viszont a hozzá tartozó OnChange eseménykezelőt indítja el.
2. A DirectoryListBox OnChange eseménykezelőjében a FileListBox.Directory tulajdonságba másoljuk át a DirectoryListBox.Directory tulajdonság értékét. Ezzel a felhasználó beavatkozása a FileListBox komponensen is látszani fog.

■ *Szöveges állományok*

A Szöveges állományok olvasható ASCII karaktereket tartalmaznak; ezek azok a fájlok, amelyeket a DOS TYPE parancsával olvashatunk el – anélkül, hogy a képernyőn zagyva jelek jelennének meg. A szöveges fájlok kezelése hasonló módon történik, mint egy kazettára való felvétel vagy lejátszás. Ebből következik, hogy a szöveges állományok bizonyos műveleteinek megvalósítása nem nagy ördöngösség, mint ahogy egy zenés műsor hangkazettára rögzítése sem az. Viszont lehetnek olyan feladatok,

amelyeket hangkazetták esetében sem könnyű kivitelezni, például a kazettán lévő egy szó megváltoztatása, vagy két kazetta tartalmának összefésülése egy kazettára. Ezek a programozónak is csak fejtörést okozhatnak.

A fölösleges munkák elkerülése céljából, csak akkor használjunk szöveges állományokat, amikor tudjuk, hogy csak tiszta szöveggel fogunk dolgozni és

- ritkán kell csak változtatnunk a fájl belsejét
- A feldolgozáskor nem kell ugrálnunk a fájlban, hanem egyesével mehetünk végig az elemeken. (Nem kell használnunk a magnó tekerőgombjait.)
- Általában csak a fájl végéhez kell dolgokat hozzáfűznünk.

Ne higgyük, hogy lehetetlen a fájl belsejét megváltoztatni, feldolgozás közben ide-oda ugrálni az elemek között vagy a fájl végétől különböző helyre beszúrni valamit. Ezek mind megvalósíthatók fáradságos munka árán.

Megjegyzés: A szöveges állomány minden sora egy kocsi vissza/soremelés karakter kombinációra végződik.

Néhány fájlkezelő függvény, mint pl. a már látott Erase, a szöveges állományok kezelését végző függvények is megkövetelik az AssignFile eljárás használatát, amellyel egy fizikai fájlt kötünk egy fájlváltozóhoz. Ebben az esetben a deklaráció így néz ki:

```
var
  FileIdentifier: TextFile;
begin
  AssignFile(FileIdentifier, filename);
```

ahol a **TextFile** kulcsszó jelzi a Delphinek, hogy szöveges fájllal dolgozunk. Ha az adott fájl nem az aktuális könyvtárban van, szükséges a fájl teljes útvonalát egyértelművé tevő információk hozzáadása a fájl nevéhez. Továbbá a fájl nevének legálisnak kell lennie.

Pascal tipp: *A régebbi Text kulcsszót csak System.Text formában használhatjuk, jobb azonban, ha egyáltalán nem használjuk.*

Az alábbi táblázat a kazettás magnóval végezhető leggyakoribb műveleteket és azok szekvenciális szövegfájl-megfelelőit tartalmazza. A szöveges állományra a TxtFile fájlváltozón keresztül hivatkozhatunk.

Magnóművelet	*Delphi megfelelő*
A készülék lejátszásra állítása és a <pillanat állj> (pause) lenyomása.	Reset(TxtFile);
A készülék felvételre állítása és a <pillanat állj> lenyomása.	Rewrite(TxtFile);
Felvétel indítása az eddig felvett rész végétől.	Append(TxtFile);

Minden alkalommal, amikor egy Reset, Rewrite vagy Append parancs kerül végrehajtásra, a Delphi felkészül az információk fogadására a fájltól illetve az információk küldésére a fájlhoz, vagy szakszerűbben: egy csatornát létesít a fájl és a program között. A Reset, Rewrite és Append utasítások után a fájl változó neve kell hogy álljon.

A CloseFile parancs kiüríti a puffert, és utasítja a DOS-t, hogy időszerűsítse az ún. állománykijelölési táblázatot (File Allocation Table – FAT). A Windows saját pufferolási technikái miatt lehet, hogy ez nem következik be közvetlenül a CloseFile parancs végrehajtása után. Emiatt egy nagyobb mértékű áramingadozás majdnem elkerülhetetlenül a nyitott fájlok adatainak elvesztéséhez vezet, ritkán még a diszkstruktúra is megrongálódhat. (Ilyenkor feltétlenül futtassuk le a CHKDSK/f vagy SCANDISK DOS parancsot.)

Megjegyzés: *A Flush eljárás segítségével a Delphit a kimeneti puffer tartalmának elmentésére kényszeríthetjük, mielőtt elkezdené a felülírását.*

Az alapértelmezés szerinti puffer 128 byte. Nagyméretű állományok feldolgozásánál ez az alkalmazás futását észrevehetően lassíthatja. A Delpiben lehetőségünk van a szöveges fájl pufferének méretét megváltoztatni. (Nagyobb puffer valószínűleg érezhetően felgyorsítja a nagy diszkigényű alkalmazásokat.)

Írás a fájlba

A Writeln parancs az egyik leggyakoribb módja annak, hogy elkülönített információkat küldjünk egy fájlba. Az alábbi példában a TEST.DAT nevű fájlt töltjük fel:

```
{Írás a fájlba}
var:
    TxtFile: TextFile;
begin
    AssignFile(TxtFile, 'TEST.DAT');
    Rewrite (TxtFile);
    Writeln (TxtFile, 'PRÓBA 1 2 3');
    CloseFile(TxtFile);
end;
```

A megszokott megjegyzés után a változó deklarációs részben létrehozzuk a TxtFile nevű, TextFile típusú változót. Az AssignFile utasítás ezt a fájlváltozót kapcsolja össze a TEST.DAT nevű állománnyal. A Rewrite utasítás megnyitja a fájlt írásra, és a fájlmutatót az állomány elejére helyezi. A Writeln utasítás adatokat juttat el a fájlba. A TEST.DAT-ban az aposztrófok közti szöveg fog megjelenni – aposztrófok nem lesznek a fájlban. A Writeln parancs a szöveget a végéhez fűzött kocsi vissza/soremelés (ASCII 13 és ASCII 10 – ezentúl „sor vége jel") kombinációval küldi el. Ez azt jelenti, hogy a fájl tartalma szavakkal a következő: a „PRÓBA" szó, vessző, szóköz, az 1 szám, szóköz, a 2 szám, szóköz a 3 szám végül egy ASCII 13 és egy ASCII 10.

Megjegyzés: Előfordul, hogy úgy akarunk írni a fájlba, hogy a sort nem zárjuk le sor vége jellel. Ekkor használjuk a Write eljárást a Writeln helyett.

Szükségünk lehet sztringlisták tartalmának (pl. a memóablak Lines tulajdonságának) fájlba másolására. A Delphiben ezt egy utasítássorral közvetlenül megvalósíthatjuk. Használjuk a TStrings objektum SaveToFile metódusát az alábbiak szerint:

```
SztringListaNeve.SaveToFile(FájlNév);
```

Adatok beolvasása szöveges állományból

Amennyiben egy fájl tartalmát egy az egyben be akarjuk másolni egy sztringlistába, nincs nehéz dolgunk – ismét elegendő egy sort írnunk.

```
SztringListaNeve.LoadFromFile(FájlNév);
```

Ha egyesével szeretnénk adatokat beolvasni egy fájlból, akkor először meg kell nyitnunk az állományt olvasásra a Reset eljárás segítségével, amelynek szintaxisa így néz ki:

Reset(fájlváltozó);

Egy megnyitott fájlból a Readln vagy a Read eljárással olvashatunk be adatokat. A Readln használatára példa az alábbi programrészlet:

```
{Olvasás fájlból}
var:
  TxtFile: TextFile;
  S: String[80];
begin
  AssignFile(TxtFile, 'TEST.DAT');
  Reset(TxtFile);
  Readln(TxtFile, S);
  CloseFile(TxtFile);
end;
```

A Readln a sztring méretének megfelelő számú karaktert olvas be – ha közben nincs sor vége jel, ami szintén a beolvasás végét jelenti – és a mutatót a következő sor elejére állítja. Ha az állomány éppen olvasás alatt álló sora hosszabb, mint a sztring hossza (jelen esetben 80), akkor a sor többi karaktere már nem kerülhet beolvasásra, mivel a következő Readln utasítás is csak a következő sor elejétől kezdene olvasni.

A Read működése egyedül abban tér el a Readln-étől, hogy nem állítja a mutatót a következő sor elejére, hanem otthagyja, ahol volt. A fenti beolvasást a Read használatával így is meg lehet valósítani:

```
{Olvasás fájlból}
var:
  TxtFile: TextFile;
  S1, S2, S3: String[5];
begin
  AssignFile(TxtFile, 'TEST.DAT');
  Reset(TxtFile);
  Read(TxtFile, S1, S2, S3);
  CloseFile(TxtFile);
end;
```

Megjegyzés: Ahogy azt az előző példa mutatja, a Read utasítás (és a Readln) argumentumában több változót is használhatunk.

I/O ellenőrzés fájlkezelés során

A valóságban nem használunk olyan programokat, amilyeneket az előző példákban láthattunk. Mindig szükséges a kivételkezelés használata, ugyanis az I/O műveletek egy olyan területét képezik a programozásnak, ahol gyakoriak a problémák. A CloseFile parancsot mindig tegyük egy **finally** blokkba, hogy elkerüljük az állománykijelölési táblázat (FAT) elrontását. Az előző rész egyik programja átírva így néz ki:

```
{Olvasás fájlból, kivételkezeléssel!}
var:
  TxtFile: TextFile;
  S: String[80];
begin
AssignFile(TxtFile, 'TEST.DAT');
try
  Reset(TxtFile); Readln(TxtFile, S);
```

```
finally;
  CloseFile(TxtFile);
end;
```

Tipp: Az EInOutError kivételt kell elfogni.

Számok olvasása szöveges állományokból

Sztringek tárolásán kívül, a szöveges fájlok számok tárolására is alkalmasak. Amikor valamilyen numerikus változót használunk a Read vagy Readln utasítás argumentumában, a Delphi automatikusan átalakítja a (számot tartalmazó) sztringet a megfelelő számmá. Precízebben, a Delphi

- átugorja a szóközöket,
- addig olvassa a számjegyeket, amíg egy szóköz, egy számjegytől különböző karakter vagy sor vége jel nem következik,
- az így kapott sztringet a megfelelő numerikus típussá alakítja, és értékül adja a numerikus változónak.

Megjegyzés: Ha az átalakítás sikertelen, akkor a Delphi egy kivételt hoz létre.

Az Eof függvény

Ha tudjuk, mennyi adat van a fájlban, akkor használhatunk egy forciklust, amellyel beolvashatjuk azokat. Ez legtöbb esetben nem túl praktikus – egyszerűen nem tudjuk hányszor menjen körbe a ciklus. Az alábbi while-ciklus váz ennek egy megkerülését mutatja be.

```
While van adat a fájlban do
  begin
    Olvasd be a következő adatot
  end;
```

Ehhez szükségünk van egy eszközre, amellyel megállapíthatjuk, hogy elértük-e a fájl végét. A Delphiben ezt az Eof (End of File=fájl vége) függvény végzi. Szintaxisa:

function Eof (var SzövegFájlAzonosító: Text): Boolean;

A függvény értéke True, ha az utolsó karaktert már beolvastuk az állományból, egyébként False. Ez azt jelenti, hogy írhatunk egy ilyen while-ciklust:

```
var
  StringVariable: String;
  TextFileID: TextFile;
begin
  AssignFile(TextFileID, 'TEST.DAT');
try
  Reset(TextFileID);
  while not Eof(TextFileID) do
    begin
      Readln(TextFileI, StringVariable);
      ShowMessage(StringVariable);
    end;
finally;
  CloseFile(TextFileID);
end;
```

Ez a program a fájl tartalmát üzenetablakokban jeleníti meg – egy sort egy ablakban.

Azért volt szükséges while-ciklust használnunk, hogy figyelembe vegyük azt a lehetőséget is, amikor a fájl ugyan létezik, de nem tartalmaz semmit. Ilyen állományokat úgy lehet előállítani, hogy írásra történő megnyitás után bezárjuk őket adatok küldése nélkül. A kódrészlet többékevésbé hű megvalósítása a fenti vázlatnak: beolvas egy adatsort (a következő sor vége jel előtt álló karaktereket, de max 255-öt), és ezt addig folytatja, míg el nem éri a fájl végét.

■ *Típusos állományok*

A *típusos* állományok egy típusból való adatokat tartalmaznak, úgymint Integer, Real vagy az előzőleg definiált Record típus. A típusos állománnyal végzett műveletek hatékonyabban végezhetők a tartalmazott adatok azonos típusából adódóan: elemei általában egyforma méretűek, és beolvasáskor nincsen szükség átalakításokra. Például számok beolvasása típusos fájlból sokkal gyorsabb, mint szöveges állományból, mert nem történik adatkonverzió – a típusos fájlok ugyanolyan formában tárolják a számokat és a sztringeket, mint maga a program. Sztringeket tartalmazó típusos állományban az elemek első byte-jában a sztring hossza szerepel; a többi byte tárolja a tulajdonképpeni sztringet. (Lehet ezek között értelmetlen karakter is, ha a sztring nem volt feltöltve a deklarált hosszig.)

A nagyobb sebesség kicsivel több programozást igényel. Az alábbi példa a típusos fájlok használatát mutatja be:

```
var
  RealFile: file of Double; {Ne használjuk a Real
                             típust, az csak kompati-
                             bilitási okokból létezik
                             a Delphiben }
  RealNumber: Double;
begin
  AssignFile(RealFile,'NUMBER.DAT');
try
  Rewrite(RealFile);
  RealNumber := Pi;
  Write(RealFile, RealNumber);
  RealNumber := Exp(1);
  Write(RealFile, RealNumber);
finally
  CloseFile(RealFile);
end;
try
  Reset(RealFile);
  Read(RealFile, RealNumber);
```

```
    Showmessage('a pi értéke: ' +
    FloatToStr(RealNumber));
  finally
    CloseFile(RealFile);
  end;
```

Általában a Write parancsot használjuk egy típusos fájlba történő íráshoz, Read paranccsal pedig az olvasást valósíthatjuk meg. (Ez utóbbit leggyakrabban a korábban már látott Eof vizsgálatot tartalmazó ciklus belsejébe illesztjük be.)

Rekordokat tartalmazó típusos állomány

A típusos fájlokban felhasználói típusokat is tárolhatunk. Az alábbi példában BookInfo típusú rekordokat használó fájlt deklarálunk:

```
type
  BookInfo = record
    Author: String[20];
    Title: String[30];
    Subject: String[15];
    Publisher: String[20];
    Miscellaneous: String[13];
  end;
var
  BookData: BookInfo;
  BookFile: file of BookInfo;
```

Ezután használhatjuk a Write és Read utasításokat rekordok fájlba történő írására, illetve onnan történő olvasásra.

■ *Típus nélküli állományok*

Ez a lehető legkisebb strukturáltságot mutató fájlfajta, akkor érdemes (szükséges) használni, ha az állomány ismeretlen, vagy gyors adatmozgatásra van szükség. Amíg a szöveges állományok jól definiálható sorokból állnak, amelyeket a sor vége jel (ASCII 13 + ASCII 10) zár le, és a típusos fájlok egy bizonyos típus alá tartozó adatokat tartalmaznak, addig a típus nélküli fájlok szinte teljesen strukturálatlanok.

Típus nélküli fájlból bármilyen típusként beolvashatunk adatokat. A programunk furcsa viselkedéséhez vezethet, ha típus nélküli állományok használatakor változóinkba nem megfelelő típusú adatokat olvasunk. Másrészt viszont a háttértár és a program közti leggyorsabb adatátvitelt ezen állományok segítségével valósíthatjuk meg.

Az alábbi programrészlet beolvas egy állományt, miközben egy másik állományba másolja.

```
var
  SrcFile: File;
  DestFile: File;
  RecsRead: Integer;
  Buffer: array [1..2048] of Byte;

begin
  AssignFile(SrcFile, 'file.in');
try
  Reset(SrcFile, 1);     {megjegyzés: a második
                         argumentum jelzi a komponens
                         (rekord vagy elem) méretet}
  AssignFile(DestFile, 'file.out');
  Rewrite(DestFile, 1); {megjegyzés: a második
                         argumentum jelzi a komponens
                         (rekord vagy elem) méretet}
  BlockRead(SrcFile, Buffer, SizeOf(Buffer),
  RecsRead);
```

```
while RecsRead > 0 do
    BlockWrite(DestFile, Buffer, SizeOf(Buffer),
    RecsRead);
    BlockRead(SrcFile, Buffer, SizeOf(Buffer),
    RecsRead);
  end;
finally
  CloseFile(SrcFile);
  CloseFile(DestFile);
end;
```

A példában sok új technikát használtunk, amelyeket érdemes közelebbről is megvizsgálni. Típus nélküli fájlok esetén a Reset és a Rewrite egy második paraméterrel is rendelkezik, amelyben a komponens (rekord vagy elem) méretét definiáljuk. Itt 1 byte-ot választottunk rekordméretnek, mivel a puffert (a Buffer nevű változót) byte-tömbként deklaráltuk – ez a lehető legprimitívebb adatszerkezet.

Típus nélküli állományoknál az adatmozgatáshoz a BlockRead és a BlockWrite eljárásokat használjuk. A paraméterek szerepe mindkét eljárás esetében megegyezik. Az első paraméter a fájlnév, a másodikban az adatszerkezet típusát adjuk meg, a harmadik a beolvasni kívánt rekordok számát kell hogy tartalmazza. A negyedik paraméter mondja meg a ténylegesen beolvasott vagy elküldött rekordok számát.

Vannak esetek, amikor szükségünk lehet az állománymutató beállítására, hogy az olvasást vagy írást attól a ponttól kezdődően végezzük. Ezt a Seek eljárással valósíthatjuk meg. Az eljárás első paramétereként a fájlváltozót adjuk meg, a második argumentumban tüntetjük fel, hogy hanyadik rekord elé álljon a fájlmutató. Szintaxisa:

Seek (ForrásFájl, Pozíció);

Tipp: Van még két I/O függvény a Delphiben, amelyekre szükségünk lehet, amikor típus nélküli fájlokkal dolgozunk. Ezek a FileSize és a FilePos. A FileSize függvény a fájl méretét adja vissza byte-okban, míg a FilePos a fájlmutató jelenlegi pozícióját mondja meg.

■ *Parancssor-paraméterek használata*

A legtöbb professzionális alkalmazás lehetővé teszi (vagy megköveteli), hogy a felhasználó a program indításakor futás közben felhasználható adatokat adjon meg: ezeket hívják *parancssor-paramétereknek*. Például, amikor azt írjuk a DOS parancssorába, hogy COPY A:*.* B: , akkor az „A:*.* B:" nem más, mint egy sztring parancssor-paraméter. A COPY parancs ezeket megköveteli, egyébként nem tudná, hogy mit másoljon hová. A Delphi támogatja az olyan alkalmazásokat, amelyek parancssor-paramétereket használnak. A szemléltetés kedvéért tegyük fel, hogy alkalmazásunkat így indítjuk a File Managerből:

```
AlkalmazásExeNév info1 info2 info3
```

A Delphi ParamCount függvényével meghatározhatjuk a megadott paraméterek számát (jelen esetben három), a ParamStr() tömb pedig a parancssor-paramétereket adja meg. A ParamStr(0) az alkalmazás elérési útvonalát és nevét tartalmazza. A ParamStr(1) az első sztringet (Info1) adja meg és így tovább.

Megjegyzés: *Nyilvánvaló, hogy fejlesztési időben nekünk is meg kell adni (ill. jó, ha megadunk) parancssor-parméter példákat, máskülönben nem lesznek adataink, és a programtesztelés nem lesz teljes. Fejlesztői környezetben ezeket a paramétereket a Run menü Parameters pontjában definiálhatjuk.*

11. FEJEZET

Windows alkalmazások közti adatcsere

A Windows 3.X és a Windows 95 is képes több alkalmazás (kvázi-)egyidejű futtatására (*multitasking*). (Azonban ennek tényleges működéséhez a Windows 3.X alatt és alkalmanként a Windows 95 vagy a Windows NT alatt is szükség van az alkalmazások és a CPU együttműködésére: az alkalmazások maguk kell hogy visszaadják a CPU használat jogát az operációs rendszernek. A Delphiben ezért az Application objektum ProcessMessages metódusa a felelős.) A multitasking akkor igazán hatékony, ha a Windows környezetben futó alkalmazások kommunikálnak egymással. Tegyük fel például, hogy egy olyan alkalmazást készítünk, amellyel követhetjük, hogy valamilyen táblázatkezelő szoftver – pl. az Excel vagy a Lotus 1-2-3 – éppen milyen tevékenységet folytat. Így aztán lehetőségünk lenne a Delphi segítségével a táblázatkezelőhöz újabb funkciókat adni. Például értesíthetnénk a felhasználót, hogy egy fontos mennyiség megváltozott. Vagy készíthetnénk egy olyan programot, amely azt vizsgálja, hogy valamilyen Windows-os szövegszerkesztőben a felhasználó hányadik sornál tart egy adott dokumentum írásakor, és ha elérte például a századik sort, akkor egy üzenetet küld a képernyőre.

Megjegyzés: A Delphi 32-ben a TThread komponens segítségével létrehozhatunk párhuzamosan futtatható programrészeket (multithreading). Ez a komponens nem tárgya a könyvnek, de annyit érdemes a multithreadigről megjegyeznünk, hogy ez a lehetőség nagymértékben különbözik a multitaskingtól, mert itt egy programon belül végezhetünk egyszerre több műveletet – például nyomtatás közben beolvashatunk a billentyűzetről –, a multitasking több program párhuzamos futását jelenti.

A legprimitívebb módja a Windows-alkalmazások kommunikációjának a DDE (dynamic data exchange = dinamikus adatkapcsolatok) használatával történik. Ez az első hallásra nagyon misztikusnak tűnő dolog elveszti varázsát, amint a Windows clipboard-jára (vágólapjára) gondolunk. A fejezet első részében a clipboarddal foglalkozunk. Ha eddig az olvasó nem szánt sok időt a clipboard tanulmányozására, akkor itt most megtudhatja, hogy az jóval több, mint szerkesztés során kivágott objektumok tárolására alkalmas hely, amelyeket később beillesztünk valahová. A DDE segítségével a programok a többi alkalmazás tudtára hozhatják, hogy mit vigyenek fel a vágólapra, illetve mit használjanak fel onnan.

Megjegyzés: A DDE-n keresztül történő kommunikáció létesítésénél kötelező valamilyen regisztrált clipboard formátum használata. Ne képzeljük, hogy a DDE ugyanúgy használja a vágólapot, mint a kivág-beilleszt művelet. Az ott található adatok nem íródnak felül, sőt lehetőség van több, egyidejű DDE kommunikációra is.

Az OLE (Object Linking and Embedding = objektumok csatolása és beágyazása) a Delphiben egy még hatékonyabb eszköz a Windows alakalmazások közti kapcsolat felállítására. Az OLE használatával egy projektben egy teljes integrált Windows alkalmazáscsomagot hozhatunk létre.

Megjegyzés: Egy Delphi 16 alkalmazásból külső programot a ShellExecute API rutinnal indíthatunk el (a szintaxist lásd a beépített helpben). A Delphi 32-ben is engedélyezett ennek a hívásnak a használata, de jobb a Win 32 API CreateProcess rutint használni.

***Visual Basic tipp:** Mivel a Delphiben nem létezik a SendKeys függvény megfelelője, billentyűleütések átirányítása más alkalmazásokhoz bizonyos Windows API rutinok rafinált használatával valósítható csak meg, de a legtöbb esetben ez is több kárt okoz, mint hasznot.*

■ *A clipboard*

A Windows clipboard alkalmas grafikus és szöveges információk Windows alkalmazások közti cseréjére, és egyes alkalmazások gyakran a kivág-beilleszt műveletekre is használják. Többek között a Delphi is a vágólapot használja a saját kivág-beilleszt szerkesztési lehetőségének megvalósítására, valamint a clipboard és a fejezet „clipboard formátumok és grafikus átvitelek" részében ismertetendő tulajdonságok segítségével projekteinket speciális funkciókkal ruházhatjuk fel.

Megjegyzés: Ahhoz, hogy a clipboardot alkalmazásainkban használhassuk, a ClpBrd unitra van szükségünk, amelynek nevét fel kell tüntetnünk a projekt ***uses*** *kulcsszavát követő felsorolásban. Ez nem vonatkozik azon alkalmazásokra, amelyek a clipboard lehetőségei közül kizárólag a kivág-beilleszt műveletet kívánják használni bizonyos szerkesztési komponensekben – mint a TEdit, TMemo, TOLEContainer, TDDEServerItem, TDBEdit, TDBImage és a TDBMemo –, amelyekbe már eleve be vannak építve a kivágást és beillesztést megvalósító metódusok.*

A Microsoft Windows felhasználók ritkán gondolnak a clipboardra, vagy ha mégis, akkor a Windows egy passzív eszközét látják benne. A Clipboard Viewer azonban egy független program, amelyet a Windows 3.X alatt általában a Windows Main programcsoportjában találhatunk, a Windows 95-ben pedig az Accessories folder alatt.

A Clipboard Viewer File menüjét használva grafikus állományokat jeleníthetünk meg, vagy a clipboard tartalmát független állományként menthetjük el. A Clipboard Viewer gyakran alkalmas arra is, hogy képeket egyik formátumból a másikba konvertáljon. A Delphi programozók is hasznát vehetik ennek a konverziós lehetőségnek, ugyanis megtehetik, hogy a Delphi forráskódot olyan formátumúvá alakítják, hogy azt más, nem ASCII alapú szövegszerkesztő, mint például a Word for Windows, is fogadni tudja.

A fejezet ezen részében a vágólapon végezhető műveletekről lesz szó, ezért érdemes a Clipboard Viewer-t megnyitni, és könnyen elérhetővé tenni. Így bármikor megtekinthetjük a clipboard tartalmát, és ellenőríz-

hetjük, hogy amit odahelyeztünk (a kódon keresztül, közvetett módon), az tényleg ott is van.

A clipboard egyszerrre csak egy adag információt képes tárolni beillesztésre és másolásra. Általában is igaz, hogy egyszerre csak egy adatcsomag tárolására alkalmas. Ha új, az előzővel megegyező formátumú információt küldünk a clipboardnak, a régi akkor is kitörlődik. Néha azonban biztosítanunk kell, hogy a clipboard üres legyen mielőtt felhasználnánk valamilyen céllal. Ezt az alábbi sorral biztosíthatjuk:

```
{Megjegyzés: a clipboard objektum a ClipBrd unitban
jön létre}
begin
  Clipboard.Clear;
end;
```

Itt az előre definiált Clipboard objektum Clear metódusát használtuk. Ha a clipboardra szöveget kívánunk küldeni, vagy onnan szöveget szeretnénk beolvasni, akkor az alábbiakban részletezett AsText metódust használjuk.

A Clipboard AsText metódusa

Az AsText metódust a következőképpen használjuk szöveg beolvasására a vágólapról:

```
StringData := Clipboard.AsText;
```

Szöveges információ küldését a clipboardra így valósíthatjuk meg:

```
Clipboard.AsText := StringData;
```

Ezzel a sorral a StringData változóban tárolt sztring a clipboardra másolódik a vágólapon előzőleg tárolt adatokat felváltva.

Clipboard formátumok és grafikus átvitelek

A clipboard felhasználásának egy másik fontos területe a grafikák tárolása. Ha egy vágólapon tárolt grafikára van szüksége programunknak, akkor letöltés előtt tudatnunk kell a Delphivel, hogy milyen formátumú kép van ott elhelyezve. Hasonlóan, amikor programból továbbítunk képeket a vágólapra, akkor a clipboardot előzetesen értesíteni kell a grafika formátumáról. Az itt közölt táblázat tartalmazza a lehetséges formátumokat és az ezeknek megfelelő konstansokat a Delphiben.

Formátum	*Konstans*
Text (.TXT)	CF_TEXT
Közönséges bittérkép (.BMP)	CF_BITMAP
Winds metafile (.WMF)	CF_METAFILEPICT
TPicture típusú objektum	CF_PICTURE
Más tartós objektum	CF_COMPONENT

A HasFormat metódus

A HasFormat metódus segítségével kérdezhetjük le, hogy a vágólapon elhelyezett grafika milyen formátumú. Használata:

Clipboard.HasFormat(*Format*)

ahol a *Format* az előző táblázat egyik konstansának értékével egyezik meg. A metódus True értékkel tér vissza, ha a vágólapon lévő kép a megadott formátumú. Például:

```
if Clipboard.HasFormat (CF_BITMAP) then ShowMessage
('Clipboard has a bitmap');
```

Egy képet a Clipboardra az Assign metódussal küldhetünk. Használata:

Clipboard.Assign(*Bitmap*);

ahol a *Bitmap* egy TGraphic, TBitmap, TMetaFile vagy TPicture típusú objektum. Ne felejtsük el: szöveget az AsText metódus segítségével tölthetünk le a vágólapról. Grafikát beolvasni a clipboardról a következő módon lehet:

1. Kérdezzük le a clipboard tartalmának formátumát.
2. Használjuk a célkomponens Assign metódusát.

Például:

```
   if Clipboard.HasFormat (CF_BITMAP) then
Image1.Picture.Bitmap.Assign(Clipboard);
```

■ *Dinamikus adatkapcsolatok (DDE)*

A DDE úgy működik, hogy egy Windows alkalmazás (elterjedt nevén kliens) egy másik Windows alkalmazást (szerver) használ bizonyos adatok megszerzésére. A DDE adatcsere gyakorlatilag két ablak között zajlik, és a Windows szempontjából a legtöbb komponens, beleértve a formokat is, ablak.

A DDE alkalmazásakor az adatcsere mindig úgynevezett párbeszéd keretében megy végbe. A Delphiben DDE szervert a DDEServerConv komponens segítségével hozahatunk létre. Annak ellenére, hogy gyakorlatilag csak a DDEServerConv komponens szolgálhat szerverként, a többi komponens is elérhetővé tehet bizonyos információkat tulajdonságain keresztül. Adatok általában a szerverből mennek a kliensbe, azonban adott esetben az információáramlás ellenkező irányba is végbemehet.

A Windows lehetővé teszi, hogy egy alkalmazás egyszerre több DDE párbeszédet is folytathat. Az sem ritka, hogy egy program egyszerre szerverként és kliensként is működik: képzeljünk el például egy Delphi projektet, amely az Exceltől kapott adatok függvényében küld információkat a Word for Windows-nak. Lényeges megkötést jelent viszont, hogy egyszerre csak egy információcsomag küldhető át.

A párbeszéd létesítéséhez szükséges, hogy ismerjük annak az alkalmazásnak a DEE nevét, amellyel fel akarjuk venni a kapcsolatot. Ha egy program támogatja a DDE-t, akkor a dokumentációjában megtalálható a DDE neve is. Például a Word for Windows-t DDE célokra „WinWord" néven lehet elérni, az Excel DDE neve ugyancsak „Excel". A Delphi

formok DDE neve az a név, amelyet .EXE fájlként adunk nekik. Ha a projektet a Delphi fejlesztői környeztéből futtatjuk, akkor a DDE név a projekt neve lesz (kiterjesztés nélkül). Ha már tudjuk annak az alkalmazásnak a nevét, amellyel kapcsolatot kívánunk létesíteni, akkor a DDEClientConv komponens DDEService tulajdonságában ezt a nevet kell feltüntetnünk.

A kapcsolat létrejöttéhez szükséges még ismernünk a DDE párbeszéd *témáját* (topic), amely általában egy speciális fájlnév. Például az Excel számára minden .XLS vagy .XLC kiterjesztésű, teljes útvonalat tartalmazó fájlnév alkalmas *témája* lehet a párbeszédnek. Végül pedig tudnunk kell, hogy éppen milyen adatokra van szükségünk (vagy milyen adatokra van a kliensnek szüksége). Ezt a DDE párbeszéd elemének (*item*) nevezzük. Például ha DDE szerverként az Excelt használjuk, akkor a párbeszéd eleme lehet egy adott cella, vagy akár több cella is egyszerre. Amennyiben TDDEServerConv komponenst használunk, akkor annak Name tulajdonsága definiálja a témát. Ennek hiányában a téma a TDDEServerItem komponenst tartalmazó form fejlécszövege lesz, ami a program futása közben változtatható, így a kapcsolatteremtés meghiúsulhat. A párbeszéd eleme a DDEServerItem komponens neve lesz.

Kilensként a TDDEClient komponens szolgál, amellyel kétféleképpen is létesíthetünk DDE kapcsolatot, a TDDEClient ConnectMode tulajdonságától függően. Ha azt DDEAutomatic-ra állítjuk, akkor a kapcsolat futási időben automatikusan létrejön, amikor a TDDEClientet tartalmazó form kinyílik. A DDEManual választással a kapcsolat csak akkor valósul meg, amikor az OpenLink metódust meghívjuk.

DDE kapcsolat létesítése fejlesztési időben

Ebben a fejezetrészben a Microsoft Excelt használjuk példaként. Arra törekedtünk, hogy a leírtak akkor is érthetőek legyenek, ha az olvasó nem rendelkezik ezzel a szoftverrel. Annyit szükséges tudni, hogy az Excelhez hasonló táblázatkezelő programokban az adatok nyilvántartása, sorok és oszlopok mentén történik, és az Excel is a Delphihez hasonlóan rendelkezik egy Edit menüvel a főmenüben, amelyben megtalálhatóak a szokásos szerkesztési műveletek köztük, a Copy is.

Nyissunk meg most egy új projektet, és adjuk hozzá a komponenspaletta System oldalán található DDEClientConv és a DDEClientItem komponenseket. Állítsuk a DDEClientItem1 DDEConv tulajdonságát DDEClientConv1-re. Indítsuk el az Excelt (vagy képzeljük azt, hogy elindítjuk az Excelt).

Létesítsünk kapcsolatot kliensként az Excellel, mint DEE szerverrel úgy, hogy a DEE párbeszéd eleme az első oszlop és az első sor találkozásánál lévő cella legyen:

1. Az Excel ablakában jelöljük ki a bal felső cellát.
2. Az Excel Edit menüjében válasszuk a Copy parancsot.
3. Menjünk vissza a Delphi ablakába, és válasszuk ki a DDEClientConv komponenst, és az objektumfelügyelőben kattintsunk a DDEService vagy a DDETopic tulajdonsághoz tartozó felsorolásjelre (...).
4. Nyomjuk le a megjelenő dialógusablak Paste Link feliratú gombját. (Ha az1–3. lépéseket helyesen követtük, akkor ezt megtehetjük, és a clipboardon rajta lesznek a kívánt DDE információk.) Kattintsunk az OK gombra.
5. Jelöljük ki a DDEClientItem komponenst. Az objektumfelügyelőben, a DDEItem tulajdonság sorában válasszuk ki az elemet a lenyíló listáról. (A clipboard tartalma nem változhat.)
6. Ha az Excel nem az aktuális könyvtárban van, akkor állítsuk be a DDEClientConv.ServiceApplication tulajdonságot az Excel elérési útvonalára. Kiterjesztés nélkül (.EXE) adjuk meg az elérhetőséget. Például: C:\EXCEL\EXCEL, ne használjunk C:\EXCEL\EXCEL.EXE-t.

Megjegyzés: Ha nem kívánjuk, hogy a szerver által szolgáltatott adatok automatikusan módosuljanak, akkor a formhoz csak a DDEClientConv komponenst adjuk hozzá. Ekkor az adatokat a szervertől kódon keresztül kérhetjük, a komponens RequestData metódusával.

Most ellenőrizzük, hogy a kapcsolatfelvétel sikeres volt-e. Menjünk vissza az Excel ablakába, írjunk be valamit a bal felső cellába, és nyomjuk le az ENTER-t. A cella tartalmának – bármi is legyen az – meg kell jelennie a DDEClientItem komponensen és annak Text tulajdonságának értékében, ugyanis a fejlesztési időben létrehozott kapcsolatok automatikusak. Ráadásul akárhányszor a DDE szerver módosítja az adatokat a

DDEClientItem számára, a Delphi kiváltja az OnChange eseményt. Ez lehetővé teszi az adatok valós idejű feldolgozását.

Ha a projektet lefuttatjuk a fejlesztési környezetből, akkor a Delphi megszakítja a DDE kapcsolatot. Az automatikus kapcsolatfelvételnél ilyenkor biztosabb a kódon keresztüli, főleg, ha egy szerver alkalmazáshoz akarunk kapcsolódni. Automatikus kapcsolódási mód bállításnál a program a fejlesztési időben létrehozott kapcsolatot próbálja visszahozni, de ez nem mindig megy.

Az így létesített kapcsolat állandó lesz. Ha elmentjük a projektet, akkor a Delphi egyes tulajdonságok értékével megőrzi a kapcsolathoz szükséges információkat. Ezekkel a tulajdonságokkal a következő fejezetrészben ismerkedünk meg. Most próbáljuk ki, hogy mi történik az előzőleg létrejött kapcsolattal, ha

1. bezárjuk az Excelt és elmentjük a projektet,
2. ideiglenesen új projektet kezdünk,
3. megnyitjuk a DDE kapcsolatot tartalmazó Delphi projektet és lefuttatjuk azt (F9).

A Delphi megkísérli elindítani a fejlesztési időben létesített DDE párbeszéd szervereként szolgáló alkalmazást.

A DDE párbeszédek lefolyását a Delphiben bizonyos tulajdonságok értékei határozzák meg. Ezen tulajdonságok értékeit programon belül manipulálhatjuk, ami a DDE párbeszédek jóval rugalmasabb használatát teszi lehetővé, mint az fejlesztési időben valaha is elképzelhető lenne.

DDE kapcsolatok futási időben

A Dephi 16 DEMOS\DDEDEMO\DDECLI.DPR példaprojekt az összes DDE-vel kapcsolatos metódust és tulajdonságot felvonultatja. A felhasználónak kell megjelölnie a DDE szerverként használni kívánt alkalmazás nevét (azaz a DDEService tulajdonság értékét), a témát és az elemet. A DDECLI projekt megnyitása után nyomógombokkal, címkékkel, szerkesztési ablakokkal, csoportablakokkal, egy DDEClientConv és egy DDEClientItem komponenssel találjuk szembe magunkat, van továbbá egy menü komponens, amellyel a kapcsolatot indíthatjuk el, és egy másik, amely az adatok áthozataláért a felelős.

Alább közöljük a kapcsolatlétesítésért felelős doNewLink és a DDEClientItemChange eseménykezelők kódját:

```
procedure TFormD.doNewLink(Sender: TObject);
begin
  DDEClient.SetLink (AppName.Text, TopicName.Text);
  DDEClientItem.DDEConv := DDEClient;
  DDEClientItem.DDEItem := ItemName.Text;
end;

procedure TFomD.DDEClientItemChange(Sender: TObject);
begin
  DDEDat.Lines := DDEClientItem.Lines;
end;
```

Ezen eseménykezelők működési elve egyszerű. A SetLink metódus létrehozza a kapcsolatot a megadott szerverrel és témával. A DDEClientItem összekapcsolódik a SetLink által létesített DDEClient párbeszéd-komponenssel. A következő sor a DDEClientItem.DDEItem tulajdonságot az elemet tartalmazó szerkesztési ablakkal köti össze. A DDECLI példaprogram azt is demonstrálja, hogyan lehet a kliensből adtatokat eljuttatni a szerverhez, makrókat (szöveges formátumú parancsokat) fogadni és lefuttatni.

DDE komponensek tulajdonságai, metódusai és eseményei

A DDE-hez nagyon sok futási időben beállítható tulajdonság tartozik. Ez a fejezetrész a legfontosabbakat mutatja be közülük. További információkat az olvasó a beépített helpből kaphat.

DDEServerItem komponens tulajdonságai

Először lássuk a DDESzerver létrehozásakor használható tulajdonságokat.

ServerConv. AServerConv tulajdonság segítségével köthetjük össze a DDEServerConv (DDE szerver párbeszéd) és a DDEServerItem (DDE

szerver elem) komponenseket. A ServerConv tulajdonság értéke a DDE párbeszédet definiáló DDEServerConv komponens neve.

Lines. A Lines tulajdonság szöveges adatokat tartalmaz a DDE kliens számára. A Lines hasonlít a Text tulajdonsághoz, az egyedüli különbség az, hogy a Text tulajdonság maximum 255 karaktert tud tárolni.

Text. A Text tulajdonság csakúgy mint a Lines, szöveges adatokat tartalmaz a DDE kliens számára. A Text tulajdonság értéke egy maximum 255 karakterből álló sztring lehet, míg a Lines tulajdonság ennél nagyobb sztringeket is tárolhat, mivel az egy sztringlista (l. az 5. fejezet sztringlistákról szóló részét).

A DDEServerItem metódusai

CopyToClipboard. A CopyToClipboard metódus a DDEServerItem Text tulajdonságában tárolt szöveges adatokat másolja a Windows Clipboardra. Ezután a DDE kliens alkalmazás aktivizálásával, a párbeszéd témájának és elemének a megjelölésével, majd az Edit | Paste Link parancs vagy annak megfelelőjének végrehajtásával létesíthetünk kapcsolatot.

A DDEServerItem eseményei

A következőkben a DDEServerItem komponensekhez kötődő legfontosabb eseményeket mutatjuk be.

OnChange. Az OnChange esemény akkor következik be, ha a DDEServerItems tulajdonságai közül valamelyik megváltozik. Ez történik, amikor az alkalmazás értékadás során megváltoztatja a Text tulajdonság értékét, vagy ha a DDE kliens, adatok küldését kezdeményezi (PokeData).

OnPokeData. Az OnPokeData eseményt az váltja ki, ha a DDE kliens alkalmazás adatokat küld a szervernek (ellentétes irányú adatátvitel).

A DDEServerConv komponens eseményei

Az alábbiakban a DDEServerConv komponensekhez kötődő legfontosabb események kerülnek bemutatásra.

Megjegyzés: A Name tulajdonságon kívül nincs más fontos DDEServerConv tulajdonság. A DDEServerConv komponens nem rendelkezik metódusokkal.

OnClose. Az OnClose esemény akkor következik be, amikor a form vagy a DDE párbeszédkomponens bezáródik.

OnExecuteMacro. Az OnExecuteMacro esemény akkor következik be, amikor a DDE kliens alkalmazás egy makrót küld a DDE szerver párbeszédkomponensnek. A makró feldolgozását egy OnExecuteMacro eseménykezelőn belül valósíthatjuk meg.

OnOpen. Az OnOpen esemény a DDE párbeszéd megnyitásakor következik be.

A DDEClientConv komponens tulajdonságai

Most röviden ismertetjük a DDEClientConv komponenshez tartozó tulajdonságokat.

ConnectMode. A ConnectMode tulajdonság értéke határozza meg a DDE szerverrel létesítendő kapcsolat típusát. A DDEAutomatic beállítással a kapcsolat automatikusan létrejön a form megnyitásakor. Ha tulajdonságnak a DDE Manual értéket adjuk, akkor a kapcsolatot a programon belül az OpenLink metódus használatával állíthatjuk fel.

DDEService. A DDEService tulajdonságban adhatjuk meg a szerver nevét. A szerverként használni kívánt alkalmazás DDE nevét az alkalmazás dokumentációjában találhatjuk meg.

DDETopic. A DDETopic tulajdonság a DDE párbeszéd témáját tartalmazza. A DDETopic-ban megadott fájlnevet a DDEService tulajdonság

értékeként megjelölt alkalmazás használja. A DDETopic megadásához szükséges lehet a szerver alkalmazás dokumentációinak tanulmányozása.

FormatChars. Ha a FormarChars tulajdonság értéke False, akkor a DDE szerver által szolgáltatott adatcsomag összes karaktere megjelenik egy az egyben a DDE kliensben. Ha értéke True, akkor a törlés (ASCII 8), a tabulátor (ASCII 9), a soremelés (ASCII 10) és a kocsi vissza (ASCII 13) nem jelenik meg a kliens oldalon.

ServiceApplication. A ServiceApplication tulajdonság tartalmazza a szerver alkalmazás végrehajtható fájljának nevét (és szükség esetén elérési útvonalát is) az .EXE kiterjesztés nélkül. Értéke általában ugyanaz, mint a DDEService tulajdonság. Néha azonban a DDEService tulajdonságban nem az .EXE fájl neve szerepel. Mivel a ServiceApplication-ban megadott név szükséges ahhoz, hogy a Delphi az inaktív DDE szervert megnyissa a kapcsolat létesítése előtt, értéke a kliensadatok között nem szerepel.

A DDEClientConv komponens eseményei

A DDE kliensekhez a Delphiben sok esemény kötődik. Ezek közül a két legfontosabbat itt ismertetjük.

OnClose. Ez az esemény akkor következik be, amikor a form vagy DDE párbeszéd bezárul.

OnOpen. Az OnOpen eseményt a DDE párbeszédek megnyitása váltja ki.

A DDEClientConv komponens metódusai

A DDEClientConv. komponenshez tartozó legyakoribb metódusokat az alábbiakban ismertetjük.

CloseLink. A CloseLink metódussal fejezhetjük be a folyó DDE párbeszédet.

ExecuteMacro. Az ExecuteMacro metódussal próbálhatunk meg egy makró sztringet küldeni a DDE szerver alkalmazásnak.

ExecuteMacroLines. Az ExecuteMacroLines metódussal próbálhatunk meg egy makró sztringlistát küldeni a szerver alkalmazásnak.

OpenLink. Az OpenLink metódussal egy új DDE párbeszédet létesíthetünk.

PokeData. A PokeData metódussal adatokat küldhetünk a DDE szerver alkalmazásnak. A metódusban max. 255 karakter hosszúságú sztringet adhatunk át.

PokeDataLines. A PokeDataLines metódussal 255 karakternél nagyobb sztringeket is elküldhetünk a szerver alkalmazásoknak.

RequestData. A RequestData metódussal adatokat kérhetünk a DDE szervertől.

SetLink. A SetLink metódusban jelölhetjük meg a DDE párbeszéd szolgáltatását (a szerver DDE nevét) és témáját. A metódus megpróbálja megnyitni a párbeszédet, ha a ConnectMode tulajdonság DDEAutomaticra van állítva. A Service paraméter definiálja a DDE szolgáltatást, amelyet a DDEService tulajdonság is megkap. A Topic paraméter a DDE témáját definiálja, amely a DDETopic tulajdonságnak is értékül adódik.

A DDEClientItem komponens tulajdonságai

DDE kliensek létrehozásakor jó néhány tulajdonságot be kell állítanunk, illetve beállíthatunk. Ezek közül itt a legfontosabbakat közöljük.

DDEConv. A DDEConv tulajdonság értéke a DDE kliens párbeszédkomponens nevét tartalmazza, amely az összeköttetés megvalósításához szükséges.

DDEItem. A DDEItem tulajdonság értéke a párbeszéd elemét határozza meg (azt, hogy pontosan milyen adat kerül átadásra a kliensnek). A szerver alkalmazás dokumentációja segítségünkre lehet e tulajdonság helyes beállításában.

Lines, Text. Mind a Lines mind a Text tulajdonság a DDE szerver által küldött szöveges adatokat tartalmazza. A különbség az, hogy a Text tulajdonságban csak 255 karaktert tárolhatunk, míg a Lines tulajdonságban, amely valójában egy sztringlista, többezer sornyi szöveget.

Megjegyzés: A DDEClientItem komponensnek nincsenek metódusai.

A DDEClientItem komponens eseményei

A DDE kliensekhez sok esemény kapcsolódik, ezek közül a legfontosabbat, az OnChange eseményt mutatjuk be.

OnChange. Az OnChange esemény akkor következik be, amikor a DDEClientItem komponens Text tulajdonsága megváltozik, amelyet a DDE szerver alkalmazás folyamatosan felfrissít.

■ *Objektumok csatolása és beágyazása (OLE)*

Az OLE (Object Linking and Embedding=objektumok csatolása és beágyazása) a dinamikus adatkapcsolatokból kifejlődött technika. Mára ez a technika messze elrugaszkodott a saját nevében hordozottaktól. Az OLE fejlettebb a DDE-nél annyiban, hogy adatok puszta átvitele mellett, az információ éppen úgy jelenik meg a célalkalmazásban, mint az eredetiben.

Az OLE-nak két verziója létezik. Az OLE 1.0 lehetővé teszi a felhasználó számára, hogy az OLE 1.0 szerver alkalmazásban létrehozott objektumot aktivizálja. Ezután a szerver alkalmazás megnyitja saját ablakát az előtérben, és átveszi a fókuszt. A tartalmazó (vagy konténer) alkalmazás egy külön ablakban fut a háttérben. Ezt *külső aktivizálásnak* (out-of-place activation) nevezik.

Az OLE 2.0-val a kimondottan OLE 2.0 szerver alkalmazásban létrehozott objektumokat aktivizálhatjuk. Ebben az esetben a szerver menürendszere és eszközpalettája a konténerkomponensben jelenik meg. Az OLE objektum szerkesztése ekkor a konténer alkalmazáson belül történik, de a műveleteket a szerver asszisztálásával végezhetjük, és annak teljes eszköztárát kihasználhatjuk. Ezt hívják *helyben történő aktivizálásnak* (in-place activation). Az OLE 2.0 gondoskodik róla, hogy a konténer alkalmazásban a táblázatok táblázatként jelenjenek meg, a dokumentumok, úgy ahogy a szövegszerkesztőben is megjelennének stb. Amikor projektünkhöz egy OLE konténerkomponenst (TOleContainer) adunk, akkor a felhasználó számára egy hidat biztosítunk egy másik Windows-alkalmazáshoz. A Delphiben sajnos nem készíthetünk OLE 2.0 szervert csak konténert.

Megjegyzés: Ahhoz, hogy egy olyan Delphi alkalmazást készíthessünk, amelyben helyben történő aktivizálás végezhető, szükséges, hogy az OLE szerver is támogassa azt. Szükséges még, hogy az alkalmazás főformjának legyen MainMenu komponense, és a projekt rendelkezzen a kapcsolódást biztosító kóddal.

Az OLE-hoz három alapfogalom tartozik. Az *objektum* az OLE szerver alkalmazásból származó összetartozó adathalmaz – például egy Excel munkalap (vagy még inkább annak egy része) – az OLE kapcsolatok tárgya.

A *csatolás* megértéséhez képzeljünk el egy csapatot, amely egy könyvön dolgozik. A szerzőn kívül sok fontos szereplő van: a műszaki szerkesztő, a lektor, a korrektor és még sokan mások. A leghatékonyabb módszer az lenne, ha a csapat minden tagja hozzáférhetne (*csatolás*) a könyvet jelentő egyetlen dokumentumhoz, és azon munkájuknak megfelelően változtatásokat végezhetnének. A dokumentum egyetlen példánya a központi szerveren lenne, így a csapat tagjainak nem kellene aggódni, hogy esetleg fontos változtatások lemaradnak a végső változatról. Más szóval, ez a modell egy párhuzamos elérési lehetőséget biztosít. Az objektum csatolása által az adatokat mindvégig a szerver alkalmazás kezeli. Gondoljunk úgy a csatolásra, mint egy lánc hozzáfűzésére egy már meglévő adathalmazhoz – az adathalmaz a lánc rázásával változtatható meg. Gyakorlatilag a csatolás során a Delphi konténer alkalmazásban a csatolt objektum adatainak egy példánya a TOleContainer komponensben tárolódik.

A *beágyazás* egy merőben más technika, amelyet általában akkor használunk, amikor több Windows alkalmazást akarunk egy kalap alá hozni. Beágyazáskor az OLE 2 a konténerkomponensre bízza az objektum adatainak teljes kezelését. Amikor a Delphi aktivizálja az OLE konténerkomponenst, a vezérlést az az alkalmazás kapja meg, amelyeben az objektum készült, amelynek lehetőségeit használva, a konténerprogramon belül módosíthatjuk az objektumot.

Megjegyzés: Létezik egy új típusú OLE, amelyet a Microsoft OLE Automationnak (OLE automatizálásnak) nevezett el. Az OLE Automation lehetőséget nyújt, hogy egy más alkalmazás nyelvét használjuk az objektum manipulálására. A Delphi 16 nem támogatja ezt a technikát, a Delphi 32 igen. (A Delphi 32-höz tartozó DEMOS könyvtárban találhatunk példaprojekteket.)

A Microsoft fő célja az OLE bevezetésével az volt, hogy a felhasználóknak módjukban álljon magának az előállítandó dokumentumnak az érdekeit szem előtt tartani, és hozzájuk igazítani a különböző szoftverek használatát. Ezzel a dokumentumcentrikus megközelítéssel sok különböző alkalmazás leghasznosabb funkciói hívhatók segítségül a dokumentum megszerkesztéséhez, ellentétben az eddigi alkalmazáscentrikus felfogással, ahol lényegében egyetlen alkalmazás állt a felhasználó rendelkezésére a kitűzött feladat megoldásához. Az OLE lehetőséget biztosít több program segítségül hívására egy adott cél eléréséhez. Tegyük fel például, hogy egy komplikált hivatalos beszámolót készítünk, amely táblázatkezelőből származó adatokat és grafikát is használ. Célszerű, ha a dokumentum egyes részeire a szövegszerkesztő felügyeljen, más részei a táblázatkezelőre legyenek bízva. Az OLE 2-ben mindig a megfelelő objektumhoz tartozó alkalmazás veszi át a vezérlést. Amikor egy objektumot beágyazunk, akkor más alkalmazás nem férhet hozzá (ellentétben a csatolással, amikor ez lehetséges). Ha a beépített objektummal dolgozni kezdünk, az objektumért felelős OLE szerver alkalmazás automatikusan elindul, akkor is, ha az addig zárva volt.

Megjegyzés: Az OLE hatékony használata komoly memóriaigényeket támaszt a hardverrel szemben. Az OLE rugalmasságát legalább 16 megabyte RAM memóriával rendelkező gépeken lehet igazán kihasználni.

Az OLE használata

A Delphiben OLE konténer alkalmazásokat a TOleContainer komponens segítségével valósíthatunk meg. Egy TOleContainer komponens formra helyezésekor úgynevezett *OLE összetett dokumentumhoz* (OLE compound document) jutunk. A komponens a komponenspaletta System oldalán található, segíthet a megtalálásban a sárga „OLE" feliratú címke.

A Delphi 16-ban kizárólag OLE kliens (konténer) projekteket hozhatunk létre, más szóval olyan programokat, amelyek a szerver alkalmazás által küldött adatok fogadására képesek. (A Delphi 1.0-hoz adott OLE konténerkomponens egy klienskomponens. Használatával OLE szerverként működő alkalmazásokat nem hozhatunk létre.)

Megjegyzés: A Delphi 32-ben lehetőség van egy speciális típusú OLE szerver, az úgynevezett OLE automation-szerver, készítésére. További információkkal a beépített help szolgál.

A most következő néhány bekezdésben az OLE-hez kapcsolódó terminológiát ismertetjük.

OLE osztályok. OLE osztályok (classes) azok az alkalmazások, amelyekből az OLE objektumok származnak. Minden alkalmazás, amelyik támogatja az OLE használatát, rendelkezik egy OLE osztálynévvel (class name) – például „WordDocument" vagy „ExcelWorksheet". (Az osztálynevekben ügyelnünk kell a kis- és nagybetűk helyes használatára.) Az elérhető osztálynevek listájához jutunk, ha az objektumfelügyelőben az ObjClass tulajdonság sorában található felsorolásjelre (...) kattintunk.

OLE dokumentumok. OLE dokumentumoknak (Documents) nevezzük az OLE objektumokat leíró adatokat tartalmazó forrásállományt. Egy OLE dokumentum lehet például Word dokumentum vagy Excel munkalap. Az elérhető osztályok listájához juthatunk, ha az objektumfelügyelőben az ObjDoc tulajdonság sorában található felsorolásjelre (...) kattintunk.

OLE elemek. OLE elemnek (items) nevezünk minden olyan adathalmazt, amelyet az OLE konténerkomponens fogadni tud. Lehet ez egy grafikon, egy táblázat néhány cellája, egy egész táblázat, szövegszerkesztett doku-

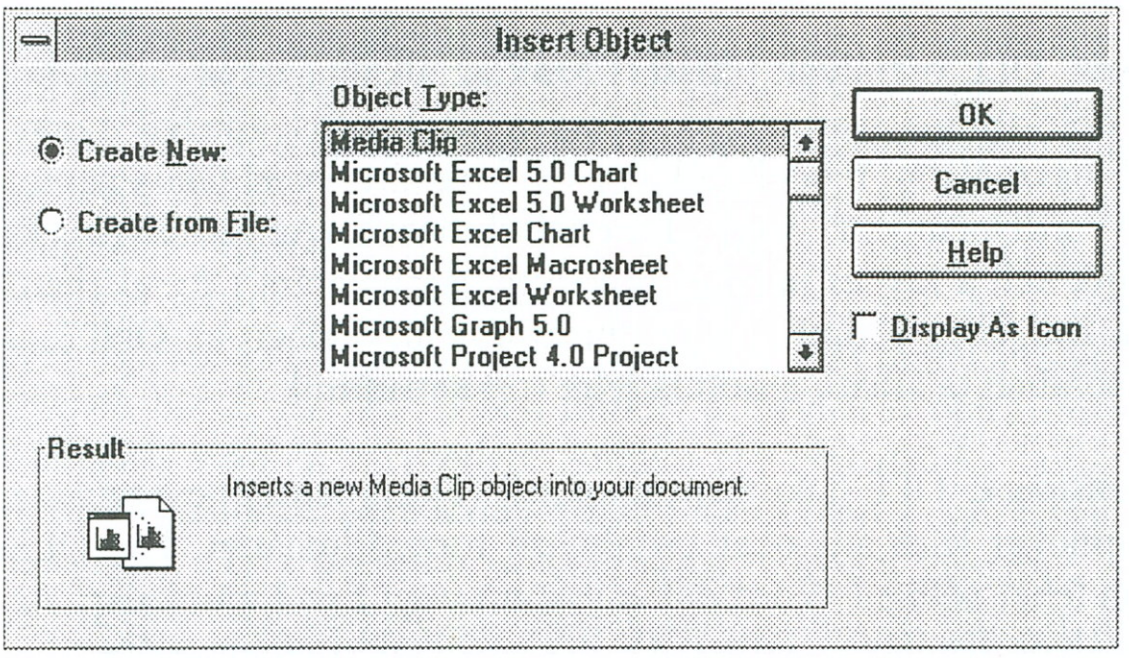
mentum egy része vagy egy teljes szövegszerkesztőben írt dokumentum. Az elérhető osztályok listájához juthatunk, ha az objektum-felügyelőben az ObjItem sorához tartozó felsorolásjelre (...) kattintunk.

■ *Az OLE 2 használata fejlesztési időben*

Az OLE 1.0-hoz képest a csatolás vagy beágyazás megvalósítása egyszerű az OLE 2-ben. Lényegében az összes munkát dialógusablakokon keresztül végezhetjük. Ezek leírásával foglalkozik ez a fejezetrész.

Ha formunkhoz hozzáadjuk a TOleContainer komponenst, és az ObjClass tulajdonság sorában található felsorolásjelre kattintunk az objektumfelügyelőben, akkor megjelenik a 11-1. ábrán látható Insert Object dialógusablak. (Ezek a Delphi 16 dialógusablakai, a Delphi 32-ben megtalálhatók sem különböznek sokban ettől.)

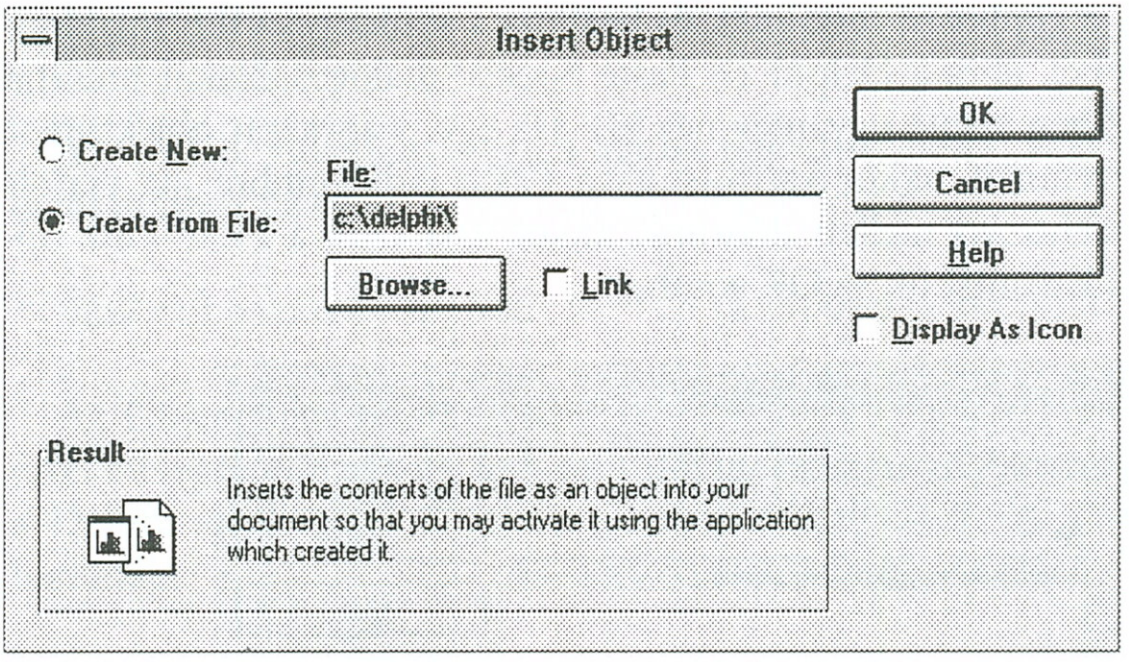
Megjegyzés: Minél több OLE-t támogató alkalmazásunk van, annál hosszabb listát kapunk.



11-1. ábra. Az OLE Insert Object dialógusablak, bekapcsolt Create New (újat készít) rádiógombbal

Az OLE Insert Object dialógusablak az összes olyan alkalmazást felsorolja, amelyet OLE szerverként használhatunk. A beágyazott vagy csatolt

objektumot megjeleníthetjük ikon formájában, vagy az OLE komponensben teljes képében. A bal oldalon található rádiógombokkal állíthatjuk be, hogy egy már kész állományt kívánunk-e fogadni (csatolt objektum) vagy egy újat akarunk készíteni (beágyazott objektum) a szerver alkalmazás segítségével. Ha az előbbi opciót (Create from File) választjuk, akkor a 11-2. ábrán látható dialógusablak jelenik meg. A Browse gombra történő kattintás után egy olyan párbeszédablakhoz jutunk, amelyben a fájlrendszerben mozogva választhatjuk ki a kívánt állományt. Ha megtettük a választást, akkor ikszeljük be a Link jelölőnégyzetet.

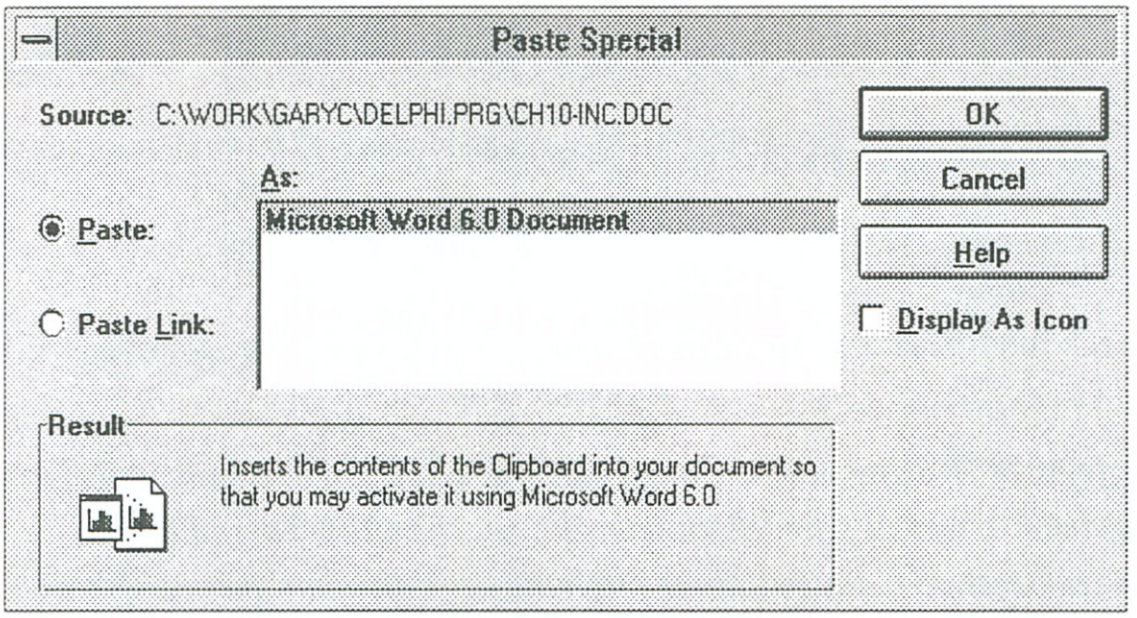


11-2. ábra. Az OLE Insert Object dialógusablak, bekapcsolt Create from File rádiógombbal

Megjegyzés: *Kattintsunk a Cancel gombra, ha az OLE tulajdonságokat programon keresztül szándékozunk beállítani. Az OLE használatához nem szükséges ezen dialógusablak megnyitása. Valójában még spórolunk is: ha egy projektet úgy fordítunk le, hogy az OLE kapcsolatot fejlesztési időben hoztuk létre, akkor a kapott .EXE fájl hosszabb lesz, mint amikor a kapcsolatot futási időben létesítjük.*

A Paste Special dialógusablak

Előfordul, hogy egy csatolt vagy beágyazott objektumot úgy kívánunk létrehozni, hogy a Windows Clipboard-on tárolt adatok alapján határozzuk meg az ObjDoc és az ObjItem tulajdonságokat. Ehhez először az alkalmazásból a vágólapra kell másolnunk az objektumot a Copy parancscsal. Ezután nyissuk meg a Paste Special dialógusablakot, amelyet fejlesztési időben úgy érhetünk el, hogy a fókuszt a TOleContainer komponensre helyezzük, majd az objektumfelügyelő ObjectItem tulajdonságának sorában a felsorolásjelre kattintunk. Ez a dialógusablak automatikusan megvizsgálja a clipboard tartalmát, és beállítja a szükséges OLE tulajdonságokat. A 11-3. ábra a Delphi 16 Paste Special dialógusablakát mutatja (a Delphi 32-ben szinte ugyanígy néz ki). Ha egy beágyazott objektumot kívánunk létrehozni, akkor kattintsunk a Paste rádiógombra; ha csatolt objektum létrehozása mellett döntünk, akkor válasszuk a Paste Link opciót, legvégül kattintsunk az OK gombra.



11-3. ábra. A Paste Special dialógusablak

A TOleContainer komponens tulajdonságai

Nem meglepő, hogy az előző dialógusablak csupán arra szolgált, hogy könnyítse az OLE komponens tulajdonságainak beállítását. A tulajdonságok beállítását azonban megtehetjük az objektumfelügyelőben is vagy a

programban (erre azért van szükség, hogy futási időben beágyazhassunk ill. csatolhassunk objektumokat).

Az AutoSize tulajdonsággal szabályozhatjuk a vezérlőelem futás alatti megjelenését. Ha értékét False-ra állítjuk, akkor a komponens saját méretétől függően vág le a megjelenítendő objektum széléből. Ha értéke True, akkor az objektum méretét a komponens automatikusan beállítja a megfelelő méretűre.

Az Insert Object dialógusablak szintén átállítja a kulcsfontosságú ObjClass tulajdonságot, amely az objektumot létrehozó alkalmazást adja meg. Például:

```
OLEContainer1.ObjClass := 'Paintbrush.Picture';
```

Az ObjDoc tulajdonságban a csatolt objektum nevét, vagy a beágyazott objektumhoz használt sablont adjuk meg. Például:

```
OLEContainer1.ObjDoc := 'C:\WINDOWS\256COLOR.BMP';
```

Az ObjectItem tulajdonságot csatolt objektumok esetén használjuk, értéke megmondja, hogy a csatolt dokumentum mely részével dolgozhat a Delphi alkalmazás. Egy táblázatrészlet megadását az alábbi példa illusztrálja:

```
OLEContainer1.ObjItem := 'R1C1:R1C10';
```

■ *OLE kapcsolatok létesítése programon keresztül*

Most, hogy túl vagyunk az OLE terminológiájával való ismerkedésen, vizsgáljuk meg a Delphi 16-hoz mellékelt DEMOS\OLE2\OLE2DEMO.DPR példaprojektet, amely a futási időben történő objektumcsatolást illusztrálja. Az OLE2DEMO projekt egy OLE2WIN nevű formot használ. A formon található egy MainMenu komponens, egy nyomógomb és egy OLE konténerkomponens. Helyben történő aktivizáláskor a főmenükomponens gondoskodik arról – a GroupIndex (csoportindex) tulajdonság beállításával –, hogy a szerver alkalmazás a saját menüjével egészíthesse ki a konténermenüt. Az OLE szerver a menüpontok három csoportját kísérelheti meg a

konténer menüjéhez fésülni. Ezek mindegyike saját csoportindex-szel rendelkezik, és bármennyi menüelemet tartalmazhat. A következő táblázatban a három menücsoport és jellemzői találhatók.

Csoport	*Index*	*Szerepkör*
Edit	1	Az aktív OLE objektum szerkesztésével kapcsolatos menüelemek, a szerver alkalmazásból.
View	3	Az OLE objektum megjelenésével kapcsolatos menüelemek, a szerver alkalmazásból.
Help	5	A szerver súgójához kapcsolódást biztosító menüelemek.

A konténer alkalmazás minden olyan menüelemét, amelynek GroupIndex tulajdonsága 1, 3 vagy 5, az OLE szerver alkalmazás hasonló csoportindexű menüeleme vált fel. A konténeralkalmazás azon menüelemeire, amelyek GroupIndex tulajdonságának értéke nem 1, 3 vagy 5, a szerver alkalmazás nincs hatással.

Megjegyzés: Lehetőség van a szerver alkalmazás eszközpalettájának és információs sorának átvételére is. Ehhez segítséget nyújt a beépített súgó vagy a Delphihez mellékelt kézikönyv 15. fejezete.

Az OLE2DEMO projekt elindítása után a File menü New pontjának választásával a form egy új példányát hozhatjuk létre, amely az alábbi kódrészlet által válik láthatóvá:

```
procedure TMainWindow.NewOleWin1Click(Sender:
TObject);
var
  Child : TOLEWin;
begin
  Child := TOLEWin.Create(Self);
  Child.Visible := True;
end;
```

Kattintsunk az Object menü Reset Object pontjára, hatására megjelenik az Insert Object dialógusablak:

```
procedure TOLEWin.ResetBtnClick(Sender: TObject);
var
  InitInfo: Pointer;
begin
  if ActiveControl.InheritsFrom (TOleContainer) then
  begin
    if InsertOleObjectDlg (Self, 0, InitInfo) = True
    then begin
      TOleContainer(ActiveControl).PInitInfo :=
      InitInfo;
      ReleaseOleInitInfo (InitInfo);
    end;
  end;
end;
```

Így egy objektumot csatolhatunk vagy ágyazhatunk be a programba. A fenti kód az OLE konténer inicializálását is megteszi az InitInfo mutató beállításával, és a mutató által lefogalalt memória felszabadításáról is gondoskodik. Az utóbbi lépésről ne feledkezzünk meg saját projektjeinkben sem.

Amikor a felhasználó bezárja az OLE konténerformot, akkor az OLE konténer Active tulajdonságnak False értéket kell adnunk a FormOnClose eseménykezelőjében.

```
procedure TOLEChild.FormClose(Sender: TObject;
var Action: TCloseAction);
begin
    OLEContainer1.Active := False;
end;
```

OLE Automation (OLE Automatizálás)

A Delphi egyik leghasznosabb tulajdonsága a bővíthetőség. Ez nem azt jelenti természetesen, hogy bárki is szívesen töltene heteket olyan saját komponensek gyártásával vagy DDL-ek készítésével, amelyek már más alkalmazásokban is (pl. Excel, Visio vagy Word) meglévő funkciókat valósítanak meg. Más (OLE-t támogató) alkalmazások nyújtotta lehetőségek kiaknázását az OLE Automation segítségével tehetjük meg. A Delphi 32-

ben írhatunk olyan programokat, amelyekkel ezen alkalmazások adatait és objektumait manipulálni tudjuk.

Megjegyzés: Az OLE Automation minden programozótól – világnézetből fakadó előítéletektől függetlenül – megköveteli a Variant típus használatát.

Néhány OLE Automationt támogató objektum a beágyazást és csatolást is lehetővé teszi. Ha egy OLE konténerobjektuma támogatja az OLE Automationt, akkor annak tulajdonságaihoz és metódusaihoz közvetlen módon hozzáférhetünk. Az OLE Automation teljes körű tárgyalása meghaladja ennek a könyvnek a lehetőségeit. Ha ez a néhány bekezdés meghozza az olvasó étvágyát, akkor érdemes elolvasni a Delphi kézikönyv ide vonatkozó részeit. Első lépésként a unit implementációs blokkjának **uses** sorát kell módosítanunk az **OleAuto** unit hozzáadásával. A kommunikációs folyamat kezeléséhez szükségünk van továbbá egy Variant típusú változóra.

Az OLE Automation használatának természetes feltétele a használni kívánt alkalmazás elindítása. A megnyitás sikertelenségét jelző hibaüzenetek elfogásáról a programban kell gondoskodni. Például a WordBasic segítségül hívásakor az alábbi kódrészletet használnánk:

```
var
  MsWordBasic: Variant;
begin
   try
    MSWordBasic := CreateOleObject('Word.Basic');
  except
   ShowMessage('A Word nem indul el - program
   befejeződik');
   Appication.Terminate;
end;
end;
```

Néhány dolgot érdemes megjegyezni mielőtt az olvasó teljesen elmélyülne az OLE Automation tanulmányozásába:

- Az OLE Automation alkalmazása megköveteli a használni kívánt objektumok teljes ismeretét. A parancsok szintaxisa eltér a megszokottól. Például az előző programrészlet által létrehozott WordBasic

objektum programozása megkívánja, hogy tisztában legyünk a WordBasic csavaros szintaxisával. Az alábbi kód egy memóablak tartalmát egy új Word dokumentumba másolja, majd kinyomtatja, elmenti és bezárja azt.

```
implementation

{$R *.DFM}

uses OleAuto;
var
   WordBasicOleAutoObject : Variant;

procedure TForm1.Button1Click(Sender: TObject);
var
  LineNo : Word;
begin
  {A FileNewDefault, Insert, FilePrintDefault,
  FileSaveAs és a FileClose metódusok leírását a
  Word 6 beépített dokumentációja tartalmazza a
  WordBasic címszó alatt.}
  try
    WordBasicOleAutoObject :=
    CreateOleObject('Word.Basic');
    WordBasicOleAutoObject.FileNewDefault;
    for LineNo := 0 to Memo1.Lines.Count - 1 do
       WordBasicOleAutoObject.Insert(Memo1.Lines
       [LineNo] + #13);
    WordBasicOleAutoObject.FilePrintDefault;
    WordBasicOleAutoObject.FileSaveAs('C:\TEST.DOC');
    WordBasicOleAutoObject.FileClose;
   except
    on E: EOleError do ShowMessage(E.Message);
  end;
end;

end.
```

- Ha OLE Automationt kívánunk használni olyan programokkal, mint az Excel vagy a Word, fel kell készülnünk arra, hogy alkalmazásunk csigalassúságú lesz. Minél nagyobb alkalmazást próbálunk irányítani, annál lassabb lesz a futás, és annál nagyobb memóriakapacitásra van szükség. Az Excelre és a Wordre illik a *nagy* jelző.

Mindig gondoljuk meg, hogy célszerű-e OLE Automationt használni nem csúcsgépeken. A memóriakapacitás még fontosabb ilyenkor, mint a CPU sebessége: még 8 megabyte RAM is kevés a boldogsághoz. Az OLE 16 megabyte-nál kezd használhatóvá válni, de 32 MB-nál (!) lesz működése kifogásolhatatlan.

Másrészt viszont egyre inkább elterjedőben vannak az OCX modellt használó kisebb és specializáltabb OLE szerverek. Például az AlphaBlox nevezetű cégnek kicsi (gyakran 300K-nál kisebb), de rendkívül hasznos OLE szerverei vannak. Ha a Microsoft (vagy a Lotus) végre eljut oda, hogy leválasztja a helyesírási elemzőt, monstre szövegszerkesztő programjának többi részétől, és külön OLE szerverként is elérhetővé teszi, akkor az is sokkal használhatóbbá válik. Az OLE Automation használata ilyen kis szerverek esetén, minden Windows 95 alatt futó gépen lehetővé válna.

Tipp: *Használjuk a Windows API rutinjait a felhasználó hardver konfigurációjának megállapítására.*

12. FEJEZET

Grafika

Ebben a fejezetben a Delphi grafikájának alapjait sajátíthatjuk el. (A grafikát irányító metódusok a Delphi mindkét verziójában ugyanazok). Rajzolhatunk formokra, nyomtatóra, vagy TImage komponensekre. A TShape komponens segítségével minimális programozással rajzolhatunk különböző geometriai alakzatokat – köröket, háromszögeket stb. A Delphi grafikai metódusainak használata legtöbbször nem kevés programkód megírását igényli, ennek fejében viszont lehetőségünk lesz a képernyőn vagy a nyomtatandó képen megjelenő minden egyes pontot ellenőrizni. A Delphi lenyűgöző grafikus lehetőségeivel olyan alkalmazásokat készíthetünk, amelyek a Microsoft Windows grafikus felületén sem vallanak szégyent.

Megjegyzés: A TPictureBox komponenssel (amelyet itt most nem tárgyalunk) bittérképeket és Windows Metafile formátumú állományokat jeleníthetünk meg. Bővebb felvilágosítást a komponensről a beépített súgó ad.

A hagyományos programozási nyelvek általában megkülönböztetést tesznek a grafikus és a szöveges megjelenítés között. Ez a Windows tisztán grafikus volta miatt majdnem teljesen elvesztette jelentőségét, ami természetesen a Delphiben is tükröződik. A Windows a szöveget is grafikusan jeleníti meg, ezért van lehetőség a sokféle font használatára, és ezért lehet szöveget pixel pontossággal elhelyezni a képernyőn a TextOut metódussal. A grafikai metódusok mindegyike a Canvas (vászon) objektumhoz tartozik és nem a formhoz vagy a képernyőhöz. Ez azt is jelenti, hogy a Form, az ImageBox, a Printer és minden olyan komponens, amelynek van Canvas objektuma képes grafika létrehozására és tárolására.

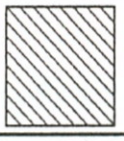
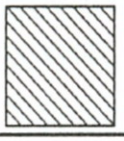
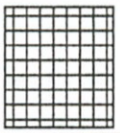
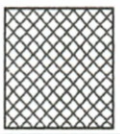
Az egyes grafikai metódusok végrehajtása után a Delphi megmondja a Windows-nak, hogy mit jelenítsen meg. A Windows feladata ezen instrukciók tolmácsolása a grafikus meghajtó számára. Ez azt jelenti, hogy a Delphi grafikus utasításainak hatása nagy mértékben attól függ, hogy a Windows milyen meghajtóprogramokat használ a képernyő és a printer vezérlésére. Ezen meghajtóprogramok használata automatikus a programozó szintjéről nézve. Nem kell azzal törődnünk, hogy a felhasználó milyen hardverkonfigurációval rendelkezik, és így a konkrét hardvertől függetlenül írjuk meg programjainkat. Ez a felfogás különbözik attól, amihez az MS-DOS programozók szoktak. Ha DOS alatt írunk grafikus programokat, akkor fontos információt jelent a program további alakulására nézve, hogy a felhasználó számítógépében milyen típusú grafikus kártya van (és hogy van-e egyáltalán).

■ *A TShape komponens*

A TShape komponens beillesztését a komponenspaletta Additional oldalán található tájképet ábrázoló gomb segítségével végezhetjük. A TShape segítségével sok különböző ábrát jeleníthetünk meg egyszerűen anélkül, hogy sok Windows erőforrást kötnénk le. A Height és Width tulajdonságok szabályozzák a geometriai alakzatokat megjelenítő komponens méretét a formon.

A TShape komponens legfontosabb része a Shape tulajdonság, amelynek értékül adható 6 különböző konstans egy-egy alakzatnak felel meg. Például az stCircle egy kört rajzol, az stRectangle egy téglalapot. Az alakzat kirajzolásának módjáért a Color és a Style tulajdonságok felelősek, amelyek a Brush tulajdonságba vannak beágyazva. A Color tulajdonság lehetséges értékeit is – a programozó munkáját könnyítve – konstansok tárolják: clWhite, clBlack és így tovább. A Style tulajdonság nyolc különböző értéket vehet fel, amelyeket ismét beépített konstansokon keresztül használhatunk. A konstansokat és a hozzájuk tartozó mintákat a 12-1. táblázatban találjuk.

12-1. táblázat. A TShape komponens Style tulajdonságának lehetséges értékei

Style tulajdonság értéke	*Minta*
bsSolid	
bsClear	
bsBDiagonal	
bsFDiagonal	
bsCross	
bsDiagCross	
bsHorizontal	
bsVertical	

■ *A grafika alapjai*

A Delphiben rajzolhatunk formra, TImage komponensre vagy a nyomtatóra. A TImage komponens működését már láttuk az első fejezeben. A TImage a komponenspaletta Additional oldalán található. Ahogy azt hamarosan látni fogjuk, mind a három objektumra ugyanúgy rajzolhatunk: a Canvas objektum segítségével. (A Canvas objektummal már találkozhattunk az 5. fejezetben.)

A TImage komponenssel megjelenített képek *tartósak*, ami azt jelenti, hogy az ablak lefedésével a kép nem törlődik le – ha a komponens újra a felszínre kerül, akkor a kép változatlanul látható lesz. A formra rajzolt

képek nem tartósak. A tartósság kérdésének memória- és sebességvetületei vannak. A tartós grafika tárolására alkalmas TImage komponens sok memóriát használ fel (a Windowsnak elegendő memóriát kell biztosítania ahhoz, hogy az egész képet tárolja, és így azt bármikor képes legyen reprodukálni.) A formra történő rajzolás kevesebb memóriát használ, de gyakran kezelése lassabb és több programozást igényel. Például ha egy valamilyen rajzot tartalmazó formot minimalizálunk, majd visszaállítjuk eredeti állapotába, akkor a kép eltűnik.

Megjegyzés: *A Delphi minden alkalommal, amikor a form egy része újra előbukkan a háttérből, aktivizálja a megfelelő OnPaint eseménykezelőt. Ezért a form (illetve a form egy részének) újrarajzolását megvalósító kódnak ott van a helye.*

A legkevésbé memóriaigényes módja annak, hogy egy form más ablakok által eltakart részeit reprodukáljuk az, ha a grafikát a form OnPaint eseménykezelőjében állítjuk vissza.

Figyelem: *Az OnPaint eseménykezelőbe ne írjunk olyan parancsokat, amelyek az adott objektumot áthelyezik vagy átméretezik. Ez ugyanis az OnPaint esemény újbóli meghívását eredményezné, ami egy végtelen ciklusba kergetné a programot.*

Grafikát tartalmazó programokban alkalmanként szükséges lehet a Refresh metódus meghívása. Ez a metódus egyaránt működik formokkal és komponensekkel. Ez a form vagy komponens azonnali felfrissítését eredményezi. A metódus végrehajtása után a Delphi meghívja az objektum OnPaint eseménykezelőjét is. Ezt a metódust sokszor a form OnSize eseménykezelő eljárásában alkalmazzuk a grafikák újra rajzolása céljából. A Delphi automatikusan frissíti a képernyőt, amikor a CPU-nak nincs más dolga, de gyakran ennek az időnek a kivárása elrondíthatja a grafikus felületet. A Component.Refresh utasítás végrehajtásával a Delphi azonnal újra rajzolja azt a komponenst, és kiváltja az OnPaint eseményt (ha az adott típusú komponenshez tartozik ilyen esemény).

A TImage komponensről részletesebben

A TImage komponens legfontosabb tulajdonságait foglaljuk most össze.

AutoSize. Ha az AutoSize tulajdonság értékét True-ra állítjuk még a kép beolvasása előtt, akkor a vezérlő elem a kép méretének megfelelően méreteződik át. Alapértéke False, ami azt jelenti, hogy a kép egy része le lesz vágva, vagy nem tölti ki teljesen a TImage komponenst.

Center. Ez a logikai tulajdonság határozza meg, hogy a komponensen megjelenő kép az ablak közepére legyen-e igazítva. Alapértéke True.

Stretch. Ha az AutoSize tulajdonságnak False, a Stretch tulajdonságnak True értéket adunk, akkor a Delphi a bittérképek és a metafájlok méretét a TImage ablakhoz igazítja, az ikonokat változatlanul jeleníti meg.

Picture. Ez a tulajdonság felelős azért, hogy milyen képet jelenítünk meg a komponensben. Általában a LoadFromFile metódussal együtt használjuk. Például így:

```
Image1.Picture.LoadFromFile('BTFLY.BMP');
```

■ *A Canvas objektum alapjellemzői*

A Canvas objektum, amely egyben a TForm, a TImage komponensek és a TPrinter objektum tulajdonsága is, kulcsszerepet játszik a Delphibeli rajzok létrehozásában. A Canvas objektum metódusaival és tulajdonságaival vihetünk végbe minden grafikai manipulációt, egy adott pont színének beállításától egészen a lekerekített téglalapok rajzolásáig.

Színek

Ha nem jelölünk ki külön színt rajzolás közben, akkor a Delphi minden grafikai metódus végrehajtásakor az aktuális objektum előtérszínét használja. (Megjegyzés: ez nem így van a TextOut, TextRect és még néhány metódus esetén, az ezek által megjelenített szöveg színét a SetColor és a SetBkColor API függvények segítségével változtathatjuk meg.) A grafikus metódus hívása előtt szükséges megjelölni a használt színt. Ezt háromféleképpen tehetjük meg:

- előre definiált színkonstansokkal, úgymint clBlack vagy clGreen,
- egy Color dialógusablakot jelenítünk meg, amelyben a felhasználó kiválaszthatja a kívánt színt, amely értékül adódik a dialógusablak Color tulajdonságának. Az ebben megadott értéket másolhatjuk be más vezérlőelemek Color tulajdonságába,
- a ColorToRGB függvény segítségével. Használata a Delphi 1.0-ban:

```
ColorToRGB(Color: TColor): Longint;
```

ahol a Color a komponens Color tulajdonsága.

Pixelgrafika

A színbeállítás után lássuk, hogyan kezelhetjük egyenként a grafika alapegységeit, a képpontokat (pixeleket). Egy képpontot a Canvas objektum Pixels tulajdonságtömbjének segítségével kapcsolhatunk be. Használata formon:

```
FormNév.Canvas.Pixels (Oszlop, Sor) := színkód;
```

TImage komponensen:

```
ImageBoxNév.Canvas.Pixels(Oszlop, Sor) := színkód;
```

Ha azon szerencsés emberek közé tartozunk, akiknek színes nyomtató is van gépükhöz csatlakoztatva, akkor a **uses** sor Printers unittal történő kiegészítése után a Pixelt így is használhatjuk:

Printer.Canvas.Pixels(*Oszlop, Sor*) := *színkód*;

A *színkód*ot helyettesíthetjük az előre definiált konstansok bármelyikével. Az első paraméter határozza meg, hogy a pont a grafikus felület melyik oszlopába, a második, hogy melyik sorába kerüljön. A fenti utasítások végrehajtásával a pont megjelenik a megadott színben és a kijelölt koordinátákon.

Alakzatok rajzolása

A Delphi lehetővé teszi, hogy néhány egyszerűbb, matematikailag jól definiált alakzatot egyetlen utasítással rajzolhassunk fel a vászonra. Ez megkönnyíti a programozó dolgát – nem kell ezekhez külön eljárásokat írnia. A Shape komponens mellett a Canvas metódusait használhatjuk erre a célra, amelyekkel egyeneseket, négyszögeket, köröket, ellipsziseket és ezek szeleteit jeleníthet meg egyetlen kódsor használatával. Az alábbi programrészlet egy olyan egyenest rajzol, amely a bal felső pontot köti össze azzal a ponttal, ahová a felhasználó kattintott az egérrel:

```
procedure TForm1.FormMouseDown(Sender: TObject;
Button: TMouseButton; Shift: TShiftState;
X, Y: Integer);
begin
  Canvas.MoveTo(0, 0);
  Canvas.LineTo(X, Y);
end;
```

Az első utasítás az egyenes kezdőpontját a (0,0) helyre, azaz a bal felső sarokba helyezi. A második utasítás felel az egyenes megrajzolásáért, a megadott paraméterek az egyenes végpontját határozzák meg – a programban az X és az Y a felhasználó által kijelölt hely két koordinátája. (Az egyenes színét a form Canvas.Brush.Color tulajdonságának aktuális értéke szabja meg.)

A Delphi kör- és ellipszisrajzoló Ellipse metódusának paramétereként a határoló téglalap egyik átlója mentén elhelyezkedő két csúcsának koordinátáit adjuk meg. A

```
Canvas.Ellipse (0, 0, ClientWidth, ClientHeight);
```

utasítással egy olyan ellipszist rajzolhatunk, amely az egész kliensterületet kitölti. Az ellipszisnek színt is adhatunk. Például a

```
Canvas.Brush.Color := clRed;
Canvas.Ellipse (0, 0, ClientWidth, ClientHeight);
```

sorok az előző ellipszist (csak a körvonalát) pirosban jelenítik meg.

Megjegyzés: *A Canvas objektum metódusaival a következő geometriai alakzatok megjelenítésére van még lehetőség: ellipszisívek, ellipsziscikkek, ellipszisszeletek, poligonok és téglalapok. Az ezeket megvalósító metódusokról a beépített súgó ad további felvilágosítást.*

A legutoljára megváltoztatott pont

A Delphi megjegyzi, hol hagyta abba a rajzolást. Ezt a helyet általában az angol szakirodalom LPR-nek (last point referenced = a pont, amelyre legutoljára hivatkoztunk) nevezi. Az LPR koordinátáit a PenPos tualjdonság tárolja.

Megjegyzés: *Ajánlatos a MoveTo metódust használni a PenPos tulajdonság megváltoztatására, és nem a direkt értékadást.*

Rajzolás tollakkal

A Canvas objektum Pen (toll) tulajdonsága szabályozza, hogy a különböző alakzatokat a metódusok hogyan rajzolják meg. A Pen tulajdonságnak 5 beágyazott tulajdonsága van: a Color, a Handle, a Mode, a Style és a Width. A toll vastagságának (Width tulajdonság) 1 pixel az alapértéke. Az alábbi sor a Width értékét változtatja meg:

```
Canvas.Pen.Width := 2;
```

Az ezután kirajzolandó alakzatok körvonala két pixel szélességű lesz.

A folytonos vonal mellett húzhatunk szaggatott, pontozott stb. vonalakat. Hét különböző stílust állíthatunk be a Pen.Style tulajdonsággal. A 12-2. táblázat sorolja fel a lehetséges értékeket.

Esetenként szükség lehet a Mode tulajdonság használatára is, amely azt határozza meg, hogy a háttér és a toll színéből hogyan álljon elő a húzott vonal színe. Ha értékének a pmNot előre definiált konstanst adjuk, akkor a Delphi a képernyő színének kiegészítő színében rajzolja majd az alakzatokat a vászonra.

12-2. táblázat. A toll Style tulajdonságának lehetséges értékei

A Style tulajdonság értéke	*A rajzolt vonal stílusa*
psSolid (alapérték)	Folytonos
psDash	Szaggatott
psDot	Pontozott
psDashDot	Vonal-pont-vonal-pont
psDashDotDot	Vonal-pont-pont-vonal-pont-pont
psClear	Láthatatlan
psInsideFrame	Zárt, határoló téglalappal rendelkező alakzatok belsejében lesz csak látható

Megjegyzés: A Canvas.Handle tulajdonság a Delphi objektumok eszközkapcsolat leíróját adja meg. Használatára akkor van szükség, amikor a Windows API függvények HDC típusú paramétert kérnek be, ilyen esetben a Canvas.Handle értékét kell átadnunk. Ezáltal a Canvas metódusaival nem megoldható problémák megoldásához igénybe vehetjük a Windows API rutinjait is.

Ecsetek használata

A Canvas objektum Brushes (ecset) tulajdonságával területeket festhetünk ki a rajzvásznon. Alapértelmezésben a téglalapok és körök belsejét a Delphi kitöltetlenül hagyja. Ezen a form vagy TPictureBox Brushes.Style tulajdonság beállításával változtathatunk. A Delphi nyolc

különböző mintázat használatát engedélyezi. Az ecset Style tulajdonsága ugyanúgy működik, mint a TShape komponensé (l. 12-1. táblázat). Ha a Style-nak például a bsSolid konstanst adjuk értékül, akkor egy kitöltött alakzatot kapunk.

Képek elmentése

A TImage komponensre rajzolt képek elmentését a Delphiben egyszerűen megtehetjük a SaveToFile utasítás segítségével. Használata:

ImageBoxNév.SaveToFile(*Állománynév*);

A komponensen lévő képet az operációs rendszer a Picture tulajdonság segítségével azonosítja. Ha a képet eredetileg egy állományból töltöttük be a TImage komponens Picture tulajdonságának értékadásával, akkor a Delphi az állományt az eredeti állomány formátumában fogja elmenteni – pl. az ikonokat ikonokként menti el. Máskülönben a Delphi bittérképként (.BMP) menti el a képet.

13. FEJEZET

Profi felhasználói felület készítése

Az előző fejezetekben elsajátíthattuk a Delphi programozás legfontosabb technikáit. Érdemes azonban néhány oldal erejéig visszatérni a felhasználói felülethez. A fejezet első részében olyan technikákat tanulhatunk, amelyek mindkét Delphi verzióban működnek. Például ebben a részben tárgyaljuk a legelterjedtebb dialógusablakok használatát. A dialógusablak megjelenése egy hajszálnyit különbözik ugyan a két verzióban, de a kezelésükhöz szükséges módszerek megegyeznek. A fejezet végén ismertetünk majd néhány komponenst, amelyeket csak a Delphi 32-ben kerültek bevezetésre. A Delphi 16-ban is találhatunk Windows 95 stílusú komponenseket, ilyen például az elegánsan kivitelezett állapotjelző sor.

Sok különböző komponens szerepeltetésekor felvetődik néhány esztétikai és funkcionális kérdés, ezek közül a legfontosabb, hogy komponensek átfedésekor mi történjen – melyik jelenjen meg fölül, és mi ezt hogyan tudjuk befolyásolni. Erről lesz szó az „Ablakok megjelenítése – Z-sorrend" fejezetrészben, a Z-sorrend szakkifejezés (angolul: Z-order), amely a komponensek mélységére utal a képernyőn; a „Z" a matematikából jön, ahol hagyományosan a z-koordinátával adjuk meg a mélységet.

A fejezet egy részét az MDI (Multiple Document Interface = több dokumentumból álló felület) alkalmazások alapjainak bemutatására szenteljük. Egy nagyobb lélegzetvételű alfejezetben megmutatjuk, hogyan lehet a felhasználó egérműveleteit nyomon követni a programon belül. Az ezt követő rész, amelynek témája a súgórendszer beillesztése alkalmazásainkba, mindkét Delphi változat használóihoz szól.

A fejezet végén a Delphi 32 Windows 95-ös vezérlőelemeiről lesz szó. A bennük rejlő lehetőségek bemutatása azonban többszáz oldalt venne igénybe, ezért itt csupán arra futotta, hogy megismerkedhessünk a Windows 95 RichTextBox (a Delplhiben TRichEdit szerkesztési komponens)

működésével, amely egy olyan TMemo-szerű komponens, amelyben több fonttípust és formátumot (aláhúzás, kiemelés stb.) használhatunk. Legvégül, mivel a világ legrosszabb helyesírói közé tartozom, nem állhatom meg, hogy ne mutassam be a Delphi 32-höz mellékelt helyesírás-ellenőrző OCX vezérlőelem használatának alapjait.

■ *Eszközpaletta és állapotjelző sor beillesztése projektjeinkbe*

A TPanel (panel) komponens segítségével helyezhetünk el legegyszerűbben eszközpalettát és állapotjelző sort mindkét Delphi változat formjain. Az Align tulajdonság használatával a panelkomponenst könnyen elhelyezhetjük a form tetejére ill. aljára. Az Align tulajdonságnak hat értéke a panel megjelenésének más-más helyet jelöl ki (l. 13-1. táblázat).

13-1. táblázat. Az Align tulajdonság lehetséges értékei

Az Align tulajdonság értéke	*Hatása*
alBottom	Ezt az értéket állapotjelző sornál használjuk. A panel a form alján jelenik meg és szélessége automatikusan a forméhoz igazodik.
alClient	Ennél a beállításnál a panel kitölti a form kliensterületét.
alLeft	A panel a form bal oldalára kerül függőleges helyzetben.
alNone	Ez az alapbeállítás, a panelt bárhová helyezhetjük a formon.
alRight	A panel a form jobb oldalára kerül függőleges helyzetben.
alTop	Ezt az értéket komponenspalettánál használjuk. A panel a form tetején jelenik meg és szélessége automatikusan a formhoz igazodik.

Eszközpaletta és állapotjelző sor létrehozása

Még az eszközpaletta elkészítése előtt fontos hogy megtervezzük, milyen ikonokat fogunk használni az egyes funkciókat megvalósító nyomógombokon. Ha ezzel megvagyunk, kövessük az alábbi instrukciókat:

1. Helyezzünk egy TPanel komponenst a formra. (A TPanel a komponenspaletta Standard oldalának végén található.)

2. A panel Height (magasság) tulajdonságát állítsuk be úgy, hogy az ikonjaink ráférjenek. (Ezt az objektumfelügyelőn keresztül vagy közvetlenül az egér használatával is megtehetjük.)
3. A panel Align tulajdonságát állítsuk alTop-ra. (Ez egyúttal a panel szélességét is a formhoz igazítja.)
4. Töröljük a form Caption tulajdonságát.
5. A panel kijelölt állapotban tartása mellett helyezzünk rá annyi TSpeedButton komponenst, amennyire szükségünk van az eszközpalettán. (Ennek a legkönnyebb módja, ha a 3. fejezetben látott „ragadós klikk" módszert használjuk.)
6. Az egyes nyomógombok Glyph tulajdonságának adjuk értékül a használni kívánt ikon nevét.

Tipp: *Professzionálisabb megjelenést kölcsönöz eszközpalettánk, ha a nyomógombokat középre igazítjuk, és lehetőleg nem hagyunk köztük felesleges üres helyeket. Ha például a panel magassága 41 a gomboké 25, akkor a SpeedButton Height tulajdonságát állítsuk 8-ra, így biztosítva, hogy a gombok az eszközpaletta közepén jelenjenek meg.*

Megjegyzés: *Azok a komponensek, amelyeket a komponenspaletta Component ikonjára történő kettős kattintással illesztünk be, automatikusan középre rendeződnek mind vízszintesen, mind függőlegesen.*

Lehetőség van a nyomógombokat olyan súgócímkékkel ellátni, amilyeneket a Delphi maga is használ. Ezáltal ha a felhasználó nincs tisztában egy gomb funkciójával, akkor az egérrel a kurzort az ikon fölé irányítja és vár, míg meg nem jelenik a szöveges eligazítás. Emlékezzünk vissza, hogy ezt a lehetőséget az adott nyomógomb (SpeedButton) ShowHint tulajdonságának True-ra állításával aktivizálhatjuk. A felbukkanó szöveget a Hint tulajdonságban adhatjuk meg, a HintPause tulajdonságban az értékének a címke megjelenése előtti várakozási időt jelölhetjük ki.

Megjegyzés: *Nyomógombcsoportok (csatolt nyomógombok) létrehozására is van mód az eszközpalettán. Megoldható például, hogy egyszerre csak egy gomb legyen lenyomva. Ezt a GroupIndex tulajdonság megfelelő beállításával érhetjük el.*

A csatolt nyomógomboknak két állapotuk van: lenyomott és felengedett. Nem csatolt gombokat is felruházhatunk ezzel a jellemzővel, ha az AllowAllUp tulajdonságot True-ra állítjuk, és a GroupIndex tulajdonságnak 0-tól különböző értéket adunk. A Down tulajdonsággal állíthatjuk be a nyomógomb kezdeti állapotát.

Állapotjelző sorok használata

Az állapotjelző sorok általában szövegorientáltak. Grafikát leginkább csak a szövegmezők elválasztásánál használunk, árnyékolt függőleges elválasztócsíkok használata változatosabbá és esztétikusabbá teszi a megjelenést. Ezeket a Canvas tulajdonság használatával illeszthetjük be. Sajnálatos módon a Panel Canvas tulajdonsága láthatatlan számunkra (private-ként van deklarálva), ezért származtatnunk kell a TPanel komponensből egy másik komponenst, amely szülője minden tulajdonságát örökli, miközben a Canvas tulajdonságot nyilvánossá teszi. Ha ezt az új objektumot használjuk, akkor grafika használata az állapotjelző soron nem okoz több nehézséget.

Az állapotjelző sort a külső események függvényében kell felfrissítenünk. A felfrissítés történhet közvetlenül a frissítést kikényszerítő esemény eljárásában. Általában jobb megoldás, ha egy külön erre írt alprogrammal módosítjuk az adott szövegmezőt (szövegmezőket) – mindig célszerű az eljárásokkal egy elemi feladatot megvalósítani, az állapotjelző frissítése ebbe már nem férne bele. (Gondoljunk csak egy dokumentum adatait megjelenítő állapotjelző sorra, a frissítő rutin feladatai közé tartozhat a dokumentum oldalainak megszámlálása, ami egy újabb hívást jelent. Ha ezzel az eseménykezelőket terheljük, a program áttekinthetősége romlik.)

Tipp: Az állapotjelző sornak még professzionálisabb külsőt kölcsönöz, ha a szöveget süllyesztve jelenítjük meg, és a panel körvonalát ízlésesen kiemeljük. Állítsuk a TPanel komponens BevelInner tulajdonságát bvLowered-re és a BorderWidth tulajdonságát 2-re.

■ *Általános dialógusablakok*

A Windows és Delphi használata során nyolc standard dialógusablak fajtát láthattunk. Ezeket használjuk állományok megnyitásakor és elmentésekor, nyomtatáskor, font választáskor, valamint színek beállításakor. Ezek a komponensek a Windows előredefiniált dialógusablakai, amelyeket a WINDOWS\SYSTEM könyvtár egy DLL-e tartalmaz (minden Windows verzióban). Ezeket a dialógusablakokat elvben könnyű használni, de a lehetőségek száma olyan nagy, hogy ez a fejezet csak éreztetni tudja programozásuk mikéntjét. Az összes általános dialógusablak megtalálható a komponenspaletta Dialogs oldalán (l. a 2. fejezetet vagy a súgócímkéket az ikonok közti eligazodáshoz). A dialógusablakok használatához a megfelelő komponenseket el kell helyezni a formon.

Meglepőnek tűnhet, hogy ezek az általános dialógusablak komponensek nem vizuálisak, azaz a program futása alatt láthatatlanok a felhasználó számára. Egy adott dialógusablak megjelenítéséhez meg kell hívnunk annak Execute metódusát. Tegyük fel például, hogy el szeretnénk érni, hogy a projekt File menüjének Open File pontjának választásakor felbukkanjon egy File Open dialógusablak. A menüponthoz tartozó (Open1OnClick) OnClick eseménykezelő eljárást ekkor így kell kezdeni:

```
procedure TForm1.Open1Click(Sender: Tobject);
begin
  if OpenDialog1.Execute then...
```

A 13-2. táblázat felsorolja a Delphiben elérhető Windows dialógusablakokat.

Megjegyzés: *Az általános dialógusablakokon át nem közvetíthetünk a felhasználó felé üzeneteket, csak a megadott célra lehet őket használni. Programon keresztül szabályozhatjuk, hogy mi történjen a felhasználó által beírt adatokkal a dialógusablak bezárása után.*

13-2. táblázat. A Delphiben elérhető általános dialógusablakok

Dialógusabalak neve	*Funkciója*
TColorDialog	Az elérhető színeket mutatja meg.
TFontDialog	Az elérhető fontokat, fontméreteket és fontszíneket tartalmazza.
TOpenDialog	Állományneveket jelenít meg. Szűrőként megadhatók benne kiterjesztések, könyvtárak és lemezmeghajtók.
TSaveDialog	Állományneveket jelenít meg. Szűrőként megadhatók itt is kiterjesztések, könyvtárak és lemezmeghajtók.
TPrintDialog	Megjeleníti a standard Windows Print dialógusablakot, amelyben a nyomtatóra küldött adatok jellemzőit állíthatjuk be. A csatlakoztatott printerről is felvilágosítást ad.
TPrinterSetupDialog	Felsorolja az elérhető nyomtatókat. Beállíthatjuk a lap orientációját és méretetét valamint a forrást. (A Print dialógusablakból is megnyitható.)
TFindDialog	Egy szerkesztési vezérlőelemet jelenítmeg, amelyben a felhasználó megadhat egy keresési sztringet.
TReplaceDialog	Ebben a dialógusablakban adhatja meg a felhasználó, hogy a dokumentum melyik sztringjét kívánja helyettesíteni és mivel.

Általános dialógusablakok kezelése

A dialógusablak megjelenítése előtt inicializálnunk kell az ablak megjelenési módjáért felelős tulajdonságokat. Például ha azt akarjuk Print dialógusablakban, alapértelmezésben csak az első oldal legyen kijelölve nyomtatásra, akkor a FromPage és ToPage tulajdonságokat ennek megfelelően kell beállítani:

```
PrintDialog1.FromPage := 1;
PrintDialog1.ToPage :=1;
```

Minden általános dialógusablak megengedi annak vizsgálatát, hogy a felhasználó rákattintott-e a Cancel gombra (vagy lenyomta-e az ESC billentyűt). Ennek figyelése majdnem minden esetben szükséges is – gondoljunk csak arra, hogy milyen törékeny volt a bittérkép-megjelenítő az első fejezetben. Az alábbi programrészlet hozzáadásával biztosíthatjuk, hogy amikor a felhasználó lenyomja az ESC billentyűt vagy a Cancel gombra kattint, akkor a változtatások ne hajtódjanak végre:

```
if OpenDialog1.Execute then
  begin
    {Az OpenDialog1.FileName értékét feldolgozó
    utasítások};
  end;
```

Ha a felhasználó bezárja a dialógusablakot a Cancel gomb vagy az ESC billentyű lenyomásával, akkor az Execute metódus False értékkel tér vissza, és így a belső utasítássorozat nem hajtódik végre. Ha az első fejezet bittérkép megjelenítőjének kódjához még egy sort hozzáírnánk, akkor nem lenne annyira sebezhető:

```
procedure TForm1.FormActivate(Sender: TObject)
begin
 if OpenDialog1. Execute then
 begin
  Image1.Picture.LoadFromFile(OpenDialog1.Filename);
 end;
end;
```

A File Open (TOpenDialog) és a File Save (TSaveDialog) dialógusablakok

A 13-3. táblázat ezen dialógusablakok legfontosabb tulajdonságait mutatja be.

Az Options tulajdonság lényeges szerepet játszik a dialógusablak megjelenésének és jellemzőinek végső kialakításában. A

```
with OpenDialog1 do
  Options := Options + [ofAllowMultiSelect];
```

programrészlet lehetővé teszi, hogy a felhasználó a dialógusablakban egyszerre több választást is megtehessen a Windows megszokott szerkesztési módszereivel. A példából kiderül az is, hogy az Options egy halmaztípusú tulajdonság. Az értékadásban az Options előzőleg beállított elemeit megtartottuk.

13-3. táblázat. A TOpenDialog és a TSaveDialog komponensek legfontosabb tulajdonságai

Tulajdonság	*Leírás*
DefaultExt	A megjelenítendő állományok kiterjesztésének alapbeállítását tartalmazza.
Title	A címsor szövegét tartalmazza. Érdemes átállítanunk, ha más szerpet szánunk a dialógusablaknak, mint a mentés vagy megnyitás.
FileName	Megadja a választott fájl nevét és elérési útvonalát.
Filter	A „List Files of Type" sort állítja be. Több szűrőt is megadhatunk ha a „|" karaktert használjuk elválasztójelként.
FilterIndex	Ezt a tulajdonságot akkor használjuk, ha a Filter tulajdonságban sok szűrőt adtunk meg.
Options	Ezzel a tulajdonsággal a dialógusablak bizonyos jellemzőit állíthatjuk be.
InitialDir	A kiindulási könyvtárat határozza meg.

Egyszerre több jellemzőt így állíthatunk be:

```
with OpenDialog1 do
  Options := Options + [ofNoChangeDir,
  ofHideReadOnly]
```

Ez a két sor az előzőleg beállított opciókat megtartva letiltja, hogy a felhasználó könyvtárat változtathasson és elrejti a Read Only jelölő négyzetet.

Amint a felhasználó megnyomja az OK gombot, ki kell hogy olvassuk a beírt vagy választott adatokat, és végre kell hajtanunk a megfelelő műveleteket. A fenti példákban az OpenDialog1.FileName tulajdonság tartalmazná a kijelölt állományt.

A TColorDialog dialógusablak

A 13-4. táblázat a TColorDialog dialógusablak legfontosabb tulajdonságait mutatja be.

Az ehhez a komponenshez tartozó szimbolikus konstansokra a „cd" előtag jellemző. A ColorDialog1.Options := ColorDialog1.Options + [cdFULLOPEN] értékadással a teljes dialógusablakot láthatóvá tesszük (azt a részt is, amelyben a felhasználó saját színeket definiálhat). Amikor

13-4. táblázat. A TColor dialógusablak legfontosabb tulajdonságai

Tulajdonság	*Leírás*
Color	A felhasználó által kiválasztott színt tartalmazza
Options	Ezen a tulajdonságon keresztül a dialógusablak bizonyos jellemzőit állíthatjuk be, működése hasonlít a TOpenDialog és a TSaveDialog komponensek Options tulajdonságához.

a felhasználó a dialógusablakból az OK gomb lenyomásával lép ki, akkor a választott szín kódját a Longint típusú ColorDialog1.Color értéke adja meg.

A TFontDialog dialógusablak

A 13-5. táblázat foglalja össze a TFontDialog dialógusablak legfontosabb tulajdonságait.

Az egyes tulajdonságok értékének kiolvasásával tudhatjuk meg, hogy a felhasználó milyen fontokat szeretne megjeleníteni. Például a FontDialog1.FontName a kiválasztott font nevét adja meg. Az adatok kiolvasása után a programban érvényt kell szereznünk a felhaszáló akaratának. Az alábbi példa az egész form fonttípusát állítja be a TFontDialog dialógusablakban megadottak alapján:

```
begin
  if FontDialog1.Execute then
  Font.Assign(FontDialog1.Font);
end;
```

13-5. táblázat. A Font dialógusablak legfontosabb tulajdonságai

Tulajdonság	*Leírás*
Font.Color	Csak színes nyomtatók esetén használható.
Font.Name	Beállítja vagy visszaadja a font nevét.
Font.Size	Beállítja vagy visszaadja a font méretét pontokban.
Options	Ezen a tulajdonságon keresztül a dialógusablak bizonyos jellemzőit állíthatjuk be, működése hasonlít a TOpenDialog és a TSaveDialog komponensek Options tulajdonságához.

A dialógusablak Options tulajdonsága halmaz típusú, amelynek 15 különböző eleme lehet, ezek mind az „fd" előtaggal kezdődnak. Az objektumfelügyelőben vagy a programban jelölhetjük ki az egyes Options elemeket. Az Option tulajdonság felelős azért, hogy mit láthatunk a Font dialógusablakban. Az fdAnsiOnly beállítással a felhasználó nem választhat szimbólumokat tartalmazó fontokat. Mint minden halmazértékű tulajdonság esetén, itt is szögletes zárójelbe tesszük a kiválasztott elemeket, és azokat vesszővel választjuk el. A beépített súgóban megtudhatjuk a többi elem szerepét.

A TPrint dialógusablak-komponens

A TPrint komponens nem más, mint a standard Windows Printer dialógusablak. Mint az előző komponenseknél, itt is az Options tulajdonság szabályozza a dialógusablak jellemzőit. A Printer dialógusablak szimbolikus konstansai „pr" vagy „po" előtaggal kezdődnek. Például, ha a PrintRange paraméternek értékül a prALLPAGES konstanst adjuk, akkor az All (összes) rádiógomb be lesz kapcsolva a Page Range (nyomtatandó oldalak) panelen. Az Options más lehetséges értékeit a beépített súgóban találhatjuk meg. A 13-6. táblázat a TPrint dialógusablak legfontosabb tulajdonságait foglalja össze.

13-6. táblázat. A TPrint dialógusablak legfontosabb tulajdonságai

Tulajdonság	*Leírás*
Copies	Beállítja vagy visszaadja a felhasználó által megadott példányszámot.
FromPage, ToPage	Meghatározza a kinyomtatandó oldalakat.
MaxPage, MinPage	Ezeken keresztül beállíthatjuk a maximális és a minimális, felhasználó által a Print Range panelen megadható oldalszámot.

■ *MDI formok*

Az MDI (multiple document interface = több dokumentumból álló felület) egy olyan lehetőség a Windows környezetben, amellyel egyetlen alkalmazás ablakában (MDI keretform vagy szülőform) egyszerre több ablakban (gyermekformok) jeleníthetünk meg szöveget, grafikát vagy más

adatokat, és azokkal párhuzamosan dolgozhatunk. Létrehozhatunk például egy MDI keretformot, amelyben a felhasználó két dokumentumot szerkeszthet egyszerre egy szövegszerkesztővel. Egy projektnek csak egy MDI szölőformja lehet. A szülőform az alkalmazás gerince, így ennek az ablaknak mindig nyitva kell lennie, ha az alkalmazás fut.

MDI szülőformot úgy hozhatunk létre, ha a projekt főformjának (amelyiket MDI keretformmá akarunk előléptetni) FormStyle tulajdonságát fsMDIForm-ra állítjuk. Az ezután a File menün keresztül hozzáadott formokat úgy tehetjük ennek gyermekformjaivá, hogy azok FormStyle tulajdonságát fsMDIChild-ra állítjuk. A már meglévő formokat is ilyen módon változtathatjuk gyermekformokká. Fejlesztési időben a gyermekablakok és a szülőablak szinte megkülönböztethetetlenek egymástól.

A projekt futása alatt azonban az összes gyermekablak a keretform belsejében lesz látható. Gondoskodnunk kell, hogy a formok Visible tulajdonsága True-ra legyen állítva.

Ha a gyermekformot a felhasználó minimalizálja, akkor az ikonként jelenik majd meg a MDI szülőformon (nem a háttérben!). Ha egy gyermekablakot maximalizálunk, akkor annak fejléce a szülőform címsorává válik. Windows API rutinok használata nélkül nem rejthetjük el, és nem is tehetjük aktivizálhatatlanná a gyermekablakokat. (Viszont létrehozhatunk újakat, és megszüntethetünk régieket.)

Az Delphi MDI formjainak egyik legnagyobb előnyeként említendő, hogy a szülőform menüi az aktív formnak megfelelően változnak. Ezáltal minden gyermekablakhoz külön menüt rendelhetünk. Az történik, hogy az aktív gyermekablakra fejlesztési időben helyezett menük futás közben a főablak menüsorában jelennek meg – a régi menüket leváltva. Így a felhasználó csak annak a gyermekablaknak a menüit láhatja a főformon, amelyre a fókuszt állította.

A Window menü – a gyermekablakok elrendezése

Minden MDI alkalmazás elengedhetetlen része a Window menü, amelyen keresztül a felhasználó elrendezheti a gyermekablakokat – hasonlóan ahhoz ahogy a Windows 3.X-ben tehetjük. A Window menüben szokták

megjeleníteni még a gyermekablakok neveit is, így a menün keresztül aktivizálhatjuk a gyermekablakokat. Tegyük fel, hogy egy MyWindows menüpont alatt szeretnénk felsorolni az elérhető gyermekformokat, ekkor a következő sort használhatjuk:

```
Form.WindowMenu := MyWindows;
```

A Delphi automatikusan beilleszti a menübe az MDI gyermekablak címsorfeliratait – és az éppen aktív form nevét kipipálja. A gyermekformok automatikus elrendezésének két módja van, az ablakok egymásra lapolása (cascade) és egymás mellé helyezése (tile). Ahhoz, hogy pl. az utóbbi funkció aktivizálható legyen a Window menüből, az alábbi kódot és annak megfelelőit kell hozzáadni a programhoz.

```
procedure TFrameForm.Tile1click(Sender: TObject);
begin
  TileMode := tbHorizontal;
  Tile;
end;
```

A példában a Tile metódust és a TileMode tulajdonságot használtuk. A másik két rendezéssel kapcsolatos metódust az alábbiakban ismertetjük.

Cascade. A Cascade metódus úgy rendezi be a keretformot a gyermekablakokkal, hogy azok átlapolják egymást, de minden ablak címsora látható marad.

ArrangeIcons. A minimalizált állapotban lévő gyermekablakok ikonjait rendezhetjük el úgy, hogy egyenletesen kitöltsék a szülőablakot.

■ *Ablakok megjelenítése – Z-sorrend*

A Delphi az alakalmazások részeit három rétegben jeleníti meg. A hátsó (alsó) rétegre a közvetlenül a formra rajzolt grafikus információk kerülnek. Ezeket a 12. fejezetben látott metódusokkal visszük a formra. A középső szinten a grafikus komponensek találhatók (egyenesek, alakzatok,

képablakok és TImage komponensek). A felső réteg a nem grafikus komponenseket tartalmazza: gombokat, jelölő négyzeteket, listaablakokat és rádiógombokat. Egyes komponensek – pl. TLabel – rendelkeznek egy Transparent nevű tulajdonsággal, amely segítségével a komponenst átlátszóvá tehetjük.

Minden rétegen belül lehetőségünk van eldönteni, hogy milyen sorrendben jelenjenek meg a komponensek. Megmondhatjuk például, hogy melyik legyen felül, ha pl. két TImage vagy nyomógomb-komponens átfedésbe kerül. MDI form használatakor pedig a gyermekablakok megjelenési sorrendjét határozhatjuk meg.

A megjelenési sorrendet a következőképpen alakíthatjuk ki: fejlesztési időben az Edit menü Bring To Front (felülre helyez) és a Send To Back (alulra helyez) opcióival beállíthatjuk a kezdeti elrendezést, futási idő alatt a BringToFront, SendToBack metódusok használatával módosíthatunk ezen.

Fontos megjegyezni, hogy egy ablak nélküli komponenst nem helyezhetünk az előtérbe, ha az egy ablakos vezérlőelem alatt van.

■ *Egérműveletek nyomonkövetése*

A Windows – és ennél fogva a Delphi is – állandóan figyeli, hogy a felhasználó mit csinál az egérrel. Egészen idáig csak két egérrel kapcsolatos eseményt használtunk, az egyik az OnClick, a másik az OnDblClick volt. Ezek jelezték, ha a felhasználó egyszer vagy kétszer egy adott formra vagy komponensre kattintott. Ebben az alfejezetben megtudhatjuk, hogyan tehetjük még árnyaltabbá az egérkezelést, és hogyan szerezhetünk tudomást egyéb egérműveletekről is. A felhasználó lenyomta az egyik egérgombot? Melyiket? A kurzor egy komponens felett áll? Felengedte a felhasználó a gombot? Melyiket? Pontosan hol található a kurzor a formon? A Delphi mindezen események észlelésére képes. Természetesen, mint mindig, ha azt akarjuk, hogy ezen eseményekre a Delphi reagáljon, akkor a megfelelő eseménykezelőkben implementálnunk kell a kívánt műveleteket. Ha például egy felbukkanó-menüt szeretnénk megjeleníteni, amikor a felhasználó lenyomja a jobb oldali egérgombot, akkor meg kell hogy írjuk azt a néhány programsort, amely ezt megvalósítja.

Fejlesztési időben a komponenseket tetszés szerint mozgathatjuk az egér segítségével. A Delphiben lehetőség van olyan alkalmazások készítésére is, amelyben a felhasználó mozgathat egyes komponenseket a Windows „fogd és vidd" technikájával. Ennek bemutatása sem marad ki ebből a részből.

Az egérrel kapcsolatos eseménykezelők

A Delphiben három alapvető egérmozgással kapcsolatos eseménykezelő eljárást használhatunk.

Eseménynév	*Mi váltja ki*
OnMouseDown	A felhasználó lenyomja az egyik egérgombot.
OnMouseUp	A felhasználó felengedi az egérgombot.
OnMouseMove	A felhasználó mozgatja az egeret.

Ezek az eljárások a billentyűzet kezelésnél látott OnKeyUp és OnKeyDown eseménykezelők megfelelői (l. 5. fejezet). A ShiftState paraméter segítségével azt is megállapíthatjuk, hogy le volt-e nyomva a SHIFT, az ALT vagy a CTRL billentyű az egérgomb lenyomásakor vagy felengedésekor.

A komponensek csak akkor ismerik fel az egérműveleteket, ha a kurzor az ő felségterületükön van. Az egérgombok valamelyikének lenyomása egy eseményt vált ki, a Delphi ezt a kurzor helyének megfelelő objektumhoz irányítja. A komponens (vagy form) egészen addig kapja az egérmozgásokról a „híreket", amíg a gombot fel nem engedi a felhasználó, ez még abban az esetben is igaz, ha a kurzor közben elhagyja a komponens (vagy form) területét.

A MouseUp és a MouseDown eseménykezelőket ugyanúgy használjuk (nem ugyanarra), és a bemenő paramétereik is megegyeznek:

```
TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
```

A MouseMove eseménykezelő eljárás paraméterei is ugyanezek leszámítva azt, hogy nincsen TMouseButton típusú (a lenyomott gomb nevét tartalmazó) argumentuma:

TForm1.FormMouseMove(*Sender*: TObject; *Shift*: TShiftState; *X*, *Y*: Integer);

Azt, hogy melyik egérgombot nyomta le a felhasználó, – ahogy azt pillanatokon belül látni fogjuk – a Button paraméter kiértékelésével állapíthatjuk meg. Hasonlóképpen, ha a felhasználó az ALT, SHIFT vagy CTRL gombok valamelyikét is lenyomta, akkor az a Shift paraméter értékében jelenik meg. Az X és az Y a kurzor pozícióját adják meg az eseménykezelőnek az objektum (form vagy képablak) belső koordinátáinak használatával, ha vannak.

Az OnMouseUp és az OnMouseDown események

Mindenekelőtt nyissunk egy új projektet; az itt közölt példákhoz szükségünk lesz rá. Menjünk az objektumfelügyelő ablakába, és az Event oldalon nyissuk meg a kódablakot az OnMouseDown eseménykezelőjéhez. Majd írjuk be a következőket:

```
procedure TForm1.FormMouseDown(Sender: TObject;
Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
Canvas.Ellipse(X-50, Y-50, X+50, Y+50);
 end;
```

Ez az egyszerű eseménykezelő az X és Y paraméterek által közvetített pozíciót használja fel. Az egér valamely gombjának lenyomásakor egy olyan kis kört rajzol, amelynek közepe a kurzor aktuális helye. Ha az alábbi OnMouseUp eseménykezelőt is hozzávesszük a projekthez, akkor az egérgomb felengedésekor a Delphi pirossal színezi ki a kör belsejét.

```
procedure TForm1.FormMouseUp(Sender: TObject;
Button: TmouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
  with Canvas do
  begin
    Brush.Style := bsSolid;     {ez a változás
                                maradandó}
    Brush.Color := clRed;       {ez az értékadás is
                                maradandó}
    Ellipse(X-50, Y-50, X+50, Y+50);
  end;
end;
```

Amennyiben a gomb lenyomása után az egeret arrébb húzzuk, egy új kört kapunk. Fontos tudnunk, hogy hiába nyomjuk le a másik egérgombot, mialatt az első még lenyomott állapotban van, a Delphi nem fog generálni egy újabb OnMouseDown eseményt, amíg az első gombot fel nem engedjük. Ez megakadályozza azt, hogy a fenti két eljárás egyes köröket kiszínezzen, míg másokat nem.

A Button paramétert akkor használnánk például, ha az lenne a célunk, hogy kiszínezett és kiszínezetlen kört is megjeleníthessünk. Tegyük fel, hogy a felhasználónak kétgombos egere van. Ekkor megtehetjük, hogy olyan eseménykezelőt írunk, amely egy kiszínezett kört rajzol, ha a felhasználó a jobb oldali egérgombot nyomja le, a bal oldali gomb lenyomásakor pedig egy színes körvonalú, de üres kört jelenít meg. A Button argumentum három különböző értéket vehet fel, amelyeknek szintén előre definiált konstansokban vannak tárolva – ezek az MbLeft, az mbRight és az mbCenter konstansok. A feladatot a fenti OnMouseUp eseménykezelő módosításával oldjuk meg:

```
procedure TForm1.FormMouseUp(Sender: TObject;
Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  with Canvas do
  begin
    Brush.Style := beSolid;
    Case Button of
    mbRight: Brush.Color := clRed;
```

```
      mbLeft: Brush.Color := clWhite;
      end;
      Ellipse(X-50, Y-50, X+50, Y+50);
    end;
  end;
```

Lehetőség van a billentyűzet és az egér együttes használatára is. Készíthetünk például egy speciális menüt, amelyet a SHIFT és a jobb oldali egérgomb együttes lenyomásával nyithatunk meg. Ehhez a Shift paramétert használó kódot kell írnunk az OnMouseDown eseménykezelőbe. A TShiftState egy halmaz típus, amelynek lehetséges elemeit a 13-7. táblázat tartalmazza.

13-7. táblázat. A TShiftState halmaztípusú paraméter lehetséges elemei

Elem	*Mit mutat?*
ssShift	A SHIFT billentyű le volt nyomva az egérgombbal együtt.
ssAlt	Az ALT billentyű le volt nyomva az egérgombbal együtt.
ssCtrl	A CTRL billentyű le volt nyomva az egérgombbal együtt.
ssRight	A jobb oldali egérgombot nyomta le a felhasználó.
ssMiddle	A közéső egérgombot nyomta le a felhasználó (általában háromgombos egér szükséges ehhez, de gyakran egy kétgombos egér mindkét gombjának egyidejü lenyomásával helyettesíthető).
ssDouble	Mindkét egérgombot lenyomta a felhasználó.

Megjegyzés: *Microsoft Windows programokban (például magában a Delphiben is) egyre jobban elterjed az a gyakorlat, hogy a jobb oldali egérgomb lenyomása egy helyzetérzékeny menüt hoz elő. Ezt az objektumfelügyelőben a felbukkanó-menü AutoPopUp tulajdonságának True-ra állításával érhetjük el a Delphiben.*

Az OnMouseMove esemény

A Delphi akkor hívja meg az OnMouseMove eseménykezelőt, amikor a felhasználó az egeret mozgatja. Nem szabad azonban azt gondolnunk, hogy az OnMouseMove esemény folyamatosan bekövetkezik, amikor a

felhasználó végighúzza a kurzort a képernyőn. Ilyen esetben a felhasználói hardver és szoftver együttesen határozzák meg az esemény bekövetkezésének gyakoriságát.

Az OnMouseMove eseménykezelő viszonylag gyakran megkapja a vezérlést, ezért a benne található program gyakran végrehajtódik, így azt érdemes a lehető legrövidebbre szabni vagy egy eljáráson belüli változóvizsgálattal biztosítani az ismételt végrehajtás elkerülését.

Az alábbiakkal gyorsíthatjuk az OnMouseMove eseménykezelő végrehajtását:

- Használjunk egésztípust számlálóként.
- Változók értékét csak akkor számoljuk újra az eljárásban, ha értéke függ az esemény paramétereitől.
- Ne közvetlenül a tulajdonságokon végezzük a manipulációkat, használjunk változókat, hogy a tulajdonságba mindig a végleges érték kerüljön.

Fogd és vidd műveletek

A Delphi projekt felhasználói felület tervezésekor a komponenseket tetszés szerint tologathatjuk a képernyőn az egér segítségével: a kurzort ráhelyezzük a mozgatni kívánt komponensre, lenyomjuk a bal oldali egérgombot, és lenyomva tartjuk, miközben az egeret úgy mozgatjuk, hogy az objektum a helyére kerüljön, végül felengedjük a gombot. Mozgás közben a komponens helyét a körvonala mutatja. Az angol szakirodalom a Microsoft által bevezetett *dragging* (húzás) kifejezést használja a komponensek ilyen mozgatására, a gomb felengedése általi pozíció felvételt *dropping*-nak (ejtés) nevezik. A magyar szakirodalom ezt fogd és vidd módszernek nevezi. Az egyszerűség kedvéért ebben a részben használjuk a „mozgatás" szót is. A Delphiben ezen lehetőségeket könnyen beépíthetjük programjainkba. Több formot használó projektjeinkben még azt is megtehetjük, hogy az egyik form objektumait átmozgathatjuk a másikba a „fogd és vidd" módszer használatával.

A komponensek kétféle mozgatást engedélyeznek: az automatikusat és a manuálisat. Ezeknek a DragMode tulajdonság két értéke felel meg. Alapértelmezésben csak speciális körülmények között helyezhetők át a komponensek. (Azt, hogy mi számít speciális körülménynek, azt mi határozhatjuk meg a programon keresztül; l. később.) Ezt hívják manuális

mozgatásnak, ekkor a DragMode tulajdonság értéke dmManual. Az érték dmAutomatic-ra változtatásával a komponenst tetszőlegesen mozgathatja a felhasználó. Függetlenül a DragMode tulajdonság értékétől, a komponens ténylegesen csak akkor fogja megváltoztani a helyét – ahogy ez a következő példából is kiderül – , ha ezt a programban is támogatjuk.

Nyissunk egy új projektet, és helyezzünk a formra egy darab nyomógombot. Állítsuk a gomb DragMode tulajdonságát dmAutomatic-ra. A mozgatást, mint egérműveletet, az OnDragDrop esemény észleli. Az esemény ahhoz a formhoz vagy komponenshez kapcsolódik, ahol az „ejtés" (drop) történik. Ha egy komponenst a formon belül kívánunk mozgatni, akkor a megfelelő kódot a form OnDragDrop eseménykezelő eljárásába kell elhelyeznünk. Ha a felhasználó számára lehetővé akarjuk tenni a nyomógomb-komponens mozgatását a formon belül, először az Accept paramétert kell True-ra állítanunk, a form DragOver eseménykezelőjében:

```
procedure TForm1.FormDragOver(Sender, Source:
TObject, X, Y
Integer, State: TDragState; var Accept: Boolean);
begin
  Accept := (Source is TButton);
end:
```

Megjegyzés: a Sender paraméter tartalmazza a form nevét, a Source a felemelt komponens nevét.

Az alábbi kód a gomb mozgatását végzi:

```
procedure TForm1,FormDragDrop(Sender, Source:
TObject; X, Y: Integer);
begin
  if Source is TButton then
  begin
    TButton(Source).Left := X;
    TButton(Source).Top := Y;
  end;
end;
```

Mivel a Source paraméter TObject típusú, először a megfelelő komponenstípussá kell konvertálnunk tulajdonságainak és metódusainak eléréséhez. A metódusokra és tulajdonságokra a szokásos ponthasználattal hivatkozhatunk – pl. TButton(Source).Left. Ha a mozgatandó komponens típusáról többet akarunk tudni, mielőtt az eljárás metódusát (metódusait) meghívjuk, akkor használjuk az **is** operátort a Source paraméterrel a 7. fejezetben leírt módon.

Futassuk le a példaprogramot. Vegyük észre, hogy a mozgatandó objektum a „fogd és vidd" művelet végrehajtása közben eredeti helyén marad. Az OnDragDrop esemény használata nem alkalmas az objektum láthatatlanná tételére a mozgatás ideje alatt. Ez azért van, mert ez az eseménykezelő csak az objektum ejtése után hívódik meg. Valójában az OnDragDrop eseménynek egyáltalán nem szükséges mozgatni a komponenst. Az eseményt gyakran arra használjuk fel, hogy a felhasználónak lehetővé tesszük benne, hogy bizonyos műveleteket kezdeményezzen, főleg akkor, amikor egyik formról kerül át valamilyen objektum egy másikra.

13-8. táblázat. A „fogd és vidd" művelettel kapcsolatos események, metódusok és tulajdonságok

Tulajdonság, esemény, metódus	*Leírás*
DragMode tulajdonság	Automatikus (dmAutomatic beállítás esetén) vagy manuális (dmManual beállítás esetén) mozgatást tesz lehetővé.
DragCursor tulajdonság	A mozgatásnál „fogd és vidd" műveletnél használt kurzort állíthatjuk be ezen keresztül.
OnDragDrop esemény	Akkor következik be, amikor a mozgatott objektumot „ráejtjük" a célkomponensre. Az esemény a célkomponenshez kapcsolódik.
OnDragOver esemény	Ha egy komponens áthalad egy másik komponens felett mozgatás során, akkor a Delphi kiváltja az utóbbi komponens OnDragOver eseményét. A komponens OnDragOver eseménykezelőben az Accept paramétert True-ra kell állítani ahhoz, hogy ledobhassuk a vitt komponenst.
OnEndDrag esemény	A „fogd és vidd művelet" befejeztével következik be.
BeginDrag metódus	Akkor használjuk, ha a DragMode dmManual-ra van állítva. Elindítja a mozgatást.
EndDrag metódus	Akkor használjuk, ha a DragMode dmManual-ra van állítva. Befejezi a mozgatást.

Mozgatáskor, alapértelmezésben, a Delphiben is használt kurzor látható a képernyőn. Ha nem ezt kívánjuk használni alkalmazásainkban, módunkban áll kicserélni azt. Ennek legegyszerűbb módja, ha már fejlesztési időben megváltoztatjuk az adott komponens DragCursor tulajdonságát, amelyet az objektumfelügyelőn keresztül érhetünk el. Itt a Delphi által felkínált kurzorokból választhatunk. Egy másik lehetőség egy „csináld magad" kurzor használata (l. 4. fejezet). Ha saját tervezésű ikont használunk, elterjedt, hogy mozgatáskor negatívban jelenítjük meg.

A 13-8. táblázat a „fogd és vidd" művelet eseményeit, metódusait és tulajdonságait foglalja össze.

Manuális és automatikus mozgatás

Ha a DragMode tulajdonság értékének dmManual alapbeállításán nem változtatunk, akkor szükséges a BeginDrag és az EndDrag metódusok használata a komponens mozgatásának engedélyezéséhez. A BeginDrag használata a következő:

component.BeginDrag(*Immediate*);

Az Immediate egy logikai paraméter, működését az alábbi táblázatból olvashatjuk ki:

component.BeginDrag(True)	Azonnal elkezdi a mozgatást.
component.BeginDrag(False)	A mozgatás akkor kezdődik, amikor az egér öt pixelnyit ment arrébb.

Ezen metódusok segítségével elérhetjük például azt, hogy a felhasználó csak akkor mozgathassa a komponenseket, ha az egérgomb mellett a CTRL billentyűt is lenyomva tartja. Ehhez az adott komponens OnMouseDown eseménykezelőjét így kell kezdenünk:

```
procedure TForm1.Button1MouseDown(Sender: TObject;
Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if ssCtrl in Shift then
```

```
    Button1.BeginDrag(True);
end;
```

Az alkalmazásainkat bemutató demóprogramokban is hasznát vehetjük ezen metódusoknak. Itt is BeginDrag metódussal indíthatjuk el a „fogd és vidd" műveletet, és az EndDrag-gal dobhatjuk le a vitt komponenst

Megjegyzés: Ha a DragMode tulajdonságot dmAutomatic-ra állítjuk, akkor a felhasználó anélkül alkalmazhatja a „fogd és vidd" módszert, hogy azt a programban engedélyeznünk kellett volna.

Az OnDragOver esemény

Minden Delphi objektum, kivéve a menüket és az időkapcsolókat, észleli ha mozgatás során felette egy másik komponens halad el. Az OnDragOver eseménykezelő használatával projektjeinknek még nagyobb rugalmasságot adhatunk. Ez az esemény lehetővé teszi a mozgatás útvonalának nyomon követését is. Használatával elérhetjük például, hogy azok a komponensek, amelyek felett a mozgatott objektum éppen tartózkodik, megváltoztatják háttérszínüket.

Az OnDragOver eseménykezelő eljárás váza formokra így néz ki:

```
procedure TForm1.FormDragOver(Sender, Source:
TObject; X, Y: Integer; State: TDragState;
var Accept: Boolean);
begin
end;
```

Komponensek esetén az eseménykezelő-váz az alábbi alakot ölti:

```
procedure TForm1.Button1DragOver(Sender, Source:
TObject; X, Y: Integer; State: TDragState;
var Accept: Boolean);
begin
end;
```

A Source a mozgatott komponens, de az eseménykezelő ahhoz a komponenshez van rendelve, amely felett a mozgatás történt. Az X és Y paraméterek a „fogd és vidd" művelet tárgyát képező komponens bal felső csúcsának a helyzetét tartalmazzák. A koordinátái, form esetén, mindig annak az objektumnak a koordinátarendszerében vannak megadva, amely fölött a form elhaladt, más komponensek esetén pedig az alul lévő formhoz viszonyítunk. A State paraméter három különböző értéket vehet fel:

A State paraméter értéke	*Jelentése*
dsDragEnter	A mozgatott objektum a célobjektum belsejében van.
dsDagLeave	A mozgatott objektum éppen elhagyta a célobjektumot.
dsDragMove	A mozgatott objektum éppen belépett a célobjektum területére.

Megjegyzés: Amíg az Accept paraméter nincs True-ra állítva, a vezérlőelemet nem lehet a célkomponensre ejteni.

■ *Súgórendszerek*

Egy professzionális alkalmazásnak nélkülözhetetlen kelléke egy olyan súgórendszer, amelyet a Windows felhasználók joggal elvárnak. Ha a beépített help-ünk nem a szokásos Windows súgórendszerek megjelenési formájához és kezeléséhez alkalmazkodik, akkor egy újabb feladatot hárítunk a felhasználóra: a súgó kezelésének megtanulását. Ezzel mi sem járunk jól, mert a külön utak választása mindig pluszmunkával jár. A Delphi jóvoltából most sem maradunk segítség nélkül, a Windows Help Compiler használatával a help-rendszer elkészítése nagyban egyszerűsödik, és a Windows-os működés miatt sem kell aggódnunk.

A Help fordító használata nagy vonalakban azon alapul, hogy a programozó megírja a szükséges szöveges állományt a megfelelő formatáló parancsokkal, amelyet a Help Compiler ugrásokká, definíciókká stb. alakít. A szükséges jelölőszavakat (mark-ups) a Delphi a beépített súgóban teszi közzé. A szöveges állományt egy olyan szövegszerkesztőben kell megírni, amely támogatja az RTF formátumot (Rich Text Format = bővített szöveges formátum). A nagyobb szövegszerkesztők közül sokan támogaják

ezt a formátumot. Érdemes azonban Windows alatt elkészíteni a helprendszer alapjául szolgáló szöveges állományt, mert ilyen módon párhuzamosan tudunk dolgozni a Help Compiler-rel – egy ablakban a szövegszerkesztőt nyitjuk meg, egy másikban a fordítót.

Mégsem ez a legkellemesebb módja a súgórendszer-készítésnek. Sok olyan külső program található, amelynek segítségével a help interaktív módon elkészíthető. Ezek gyakran megérik az árukat.

Tipp: A kedvenc help-rendszer építő segédprogramom a Blue Sky Software által készített RoboHelp, amelynek Windows 3.X és Windows 95 alatti verziója egyaránt létezik. A Microsoft Software Developers CD-ROM-on más jónevű szoftverek is találhatók erre a célra: a WHAT (Windows Help Authoring Templates) és a WHPE (Windows Help Project Editor). További eszközökről adhat felvilágosítást a CompuServe WINDISK fórumának WinHelp Lib szekciója.

Arra is ügyelnünk kell, hogy ne maradjon el a felhasználók által megszokott, biztos pontnak számító Help menü. A Help menü kell hogy tartalmazza a Contents (Tartalom), Search (Keresés) és az About (Névjegy) pontokat. A Search menüpontnak kell biztosítania, hogy a felhasználó az alkalmazás kulcsszavai között keressen. Ezeken a kulcsszavakon keresztül kapcsolódhat majd a különböző témakörökhöz, amelyeket előzetesen megírtunk. Az alkalmazás egyes részeit célszerű ellátni helyzetérzékeny súgóval. Ilyen módon a felhasználó az F1 lenyomásával kaphat felvilágosítást attól függően, hogy a felhasználói felület melyik részén tartózkodik.

A TApplication objektum HelpFile tulajdonságán keresztül rendelhetünk egy (lefordított) Help állományt az alkalmazáshoz. A formok és komponensek HelpContext tulajdonságának beállításával a helyzetérzékeny súgót „helyezhetjük üzembe". A HelpContext tulajdonságok értékei mondják meg a Help Compiler-nek a helyzetérzékeny súgó csatlakozási pontjait az egyes témákhoz.

A Help Compiler munkájához szükséges fontos információkat a Help projekt állomány tartalmazza. Ez mondja meg a fordítónak, hogyan építse fel a Help fájlt a különböző témákat tartalmazó fájlokból (topic file) és a bittérképeket és metafile-okat tartalmazó grafikus állományokból. A Help projekt fájl egy közöséges ASCII állomány, amelynek .HPJ kiterjesztést

kell adnunk. Lefordítás után a fordító ebből egy .HLP kiterjesztésű állományt készít. A Help project fájlban található a Help témáit leíró tartalomsztringek (context strings) és az alkalmazás által, a helyzetérzékeny súgó aktivizálásakor átadott tartalomszámok (context numbers) közti megfeleltetési táblázat. Arra is lehetőségünk van, hogy két tartalomsztringet ugyanahhoz a témához rendeljünk.

A tartalomsztringek és tartalomszámok megfeleltetését a [MAP] kulcsszó után tehetjük meg. A témamegjelölést üres hely (szóköz vagy tabulátor) kell hogy kövesse, majd a megfelelő tartalomszám. Az alábbi néhány sor bemutatja, hogyan néz ki egy Help projekt fájl [MAP] kulcsszóhoz tartozó része:

```
[MAP]
FILE_MENU      5     ;5 a tartalomszámot jelenti
                     ;Megjegyzéseket a pontosvessző
                      után lehet írni
EDIT_MENU     10     ;10 a tartalomszámot jelenti
VIEW_MENU     15     ;15 a tartalomszámot jelenti
```

Ezek után ha a Windows help-kezelőjét (Windows help engine) egy olyan komponensből hívjuk meg (pl. az F1 lenyomásával), amelynek HelpContext tulajdonsága 5, akkor a help-rendszer FILE_MENU oldala fog megjelenni. A lényeg az, hogy a tartalomszám a HelpContext tulajdonság értékeként adódik át.

Tipp: *Ha szeretnénk, hogy az általános dialógusablakokban megjelenjen a Help nyomógomb, akkor a dialógusablak Options tulajdonságához hozzá kell vennünk a ShowHelp elemet.*

A Windows help-kezelő meghívása

A TApplication objektum három olyan metódussal rendelkezik, amellyel a Windows help-kezelőt meghívhatjuk. Mind a három metódus az aktuális operációs rendszernek megfelelő WinHelp API függvényt hívja meg. (Természetesen a WinHelp API függvényt közvetlenül is használhatjuk.) Ezek mindegyike egy logikai értéket ad vissza, amely azt mondja meg, hogy sikeres volt-e a hívás.

HelpCommand. Ez a metódus lényegében a WinHelp API rutinnal ekvivalens. Ennek megfelelően a paraméterei is majdnem ugyanazok, mint a WinHelp függvénynek. (Ezek megnézhetők a beépített súgóban.) Szintaxisa:

function HelpCommand(*Command*: Word; *Data*: LongInt): Boolean;

A Data paraméter csak a Delphi 16-ban LongInt típusú, a Delphi 32-ben Integer. A Command paraméter funkciója megegyezik a WinHelp API függvény wCommand paraméterével. Például amelyek a User vagy User 32 könyvtárban találhatók. (User.exe és User 32 dll)

```
Application.HelpCommand(HELP_HELPONHELP, 0);
{vagy használjunk a 4-et}
```

sor a „Help on Help" oldalt jeleníti meg.

Figyelem: A HelpCommand csak akkor működik, ha az Application.HelpFile tulajdonság értékét beállítottuk úgy, hogy az a megfelelő súgóállományt tartalmazza.

HelpContext. A függvény szintaxisa egyszerű:

function HelpContext(Context: THelpContext): Boolean;

A súgó tartalomszámát kell megadnunk a Context paraméter értékeként, hatására a súgó megfelelő oldala jelenik meg a képernyőn. (A tartalomszámok elvben lehetnek negatívak is, ezért a THelpContext típus megengedi a pozitív és negatív számok használatát is a Delphi 16 LongInt illetve a Delphi 32 Integer határáig.)

HelpJump. Ez a megfelelő ugrást hajtja végre. Szintaxisa:

function HelpJump(const JumpId: string): Boolean;

A sztringben az ugrás nevét kell megadni.

■ *Windows 95 vezérlőelemek*

A Delphi hozzáférést biztosít egyes Windows 95-ös vezérlőelemek számára is. Ezeket az alábbi táblázat foglalja össze.

Megjegyzés: *A Delphi 16 is messzire jutott a Windows 95 megjelenésű általános vezérlőelemek implementálásában. Ezért sok Delphi programozó olyan vizuális komponensek használatát részesíti előnyben, amelyek mindkét Delphi változatban megtalálhatók.*

TTabControl	Nagyon hasonló a TTabSet komponenshez – éppen ezért ritkán használt.
TPageControl	Ezzel a Delphiben is használt, több oldalas dialógusablakokat készíthetünk.
TTreeView	Vázlatok megjelenítéséhez használható.
TListView	Több oszlopos lista készítéséhez használható.
TImageList	Ez a komponens a TTreeView vezérlőelem képeit tárolja. (A TTreeView komponens Image tulajdonságában megadott képek is megjelennek a hozzá tartozó formon éppen úgy, mint a TImageList komponensek.)
TTrackBar	Lineáris potméter alakú görgetősáv.
TProgressBar	Ez a vizuális komponensfolyamatok állapotát jelzi. A TGauge komponens hasonlóan működik, de rugalmasabb, ezért ezt a komponenst sem használjuk gyakran.
THeaderControl	Ezzel a komponenssel mozgatható fejléceket hozhatunk létre, hasonló a THeader komponenshez.
TStausBar	Állapotjelző sorok készítéséhez használható.
TRichEdit	Olyan szerkesztési komponens, amelyben több fonttípust is használhatunk.
TUpDown	Felfele és lefele nyilak, értékek növelésére és csökkentésére.
THotKey	Egy speciális szerkesztési ablak. Olyan dialógusablakokban használjuk, amelyekben forróbillentyűket definiálhatunk.

A TRichEdit komponens

A TRichEdit komponens egyike a leghasznosabb Win 95-ös vezérlőelemeknek. Lehetőségei meghaladják a közönséges szerkesztési- és memókomponensekét. A TRichEdit komponens támogatja több fonttípus egyidejű használatát. Tulajdonságai zöme megegyezik a TMemo komponensével. A jól ismert TString típusú Lines tulajdonság itt is a szöveg sorait tárolja.

Megjegyzés: A TRichEdit komponensben a formatálás legtöbbször a kijelölt részeket érinti. Mint más szerkesztési vezérlőelemnél, a kijelölést a SelText tulajdonsággal olvashatjuk ki, vagy változtathatjuk meg. Két másik idevágó fontos tulajdonság a SelLength és a SelStart.

A következő táblázat a TRichEdit legfontosabb, a régi szerkesztési komponensekben nem megtalálható tulajdonságait tartalmazza:

Tulajdonság	*Funkció*
DefaultText	Megszabja, hogyan jelenjen meg a kiindulási szöveg.
PlainText	Ez a logikai tulajdonság azt mondja meg, hogy a komponens tartalma közönséges szövegként vagy bővített szövegként (rich) kezelendő-e.
SelAttributes	A TRichEdit kulcsfontosságú tulajdonsága, amely a szöveg megjelenítési módjáért felelős. Ez egy objektum saját tulajdonságokkal, úgymint Color, Name, Pitch, Size, Style és Height. Ezek működése megegyezik a TFont objektum tulajdonságainak működésével. A SelAttributes tulajdonság fontjellemzőivel felruházhatjuk az éppen kijelölt szöveget vagy fordítva, a kijelölt szöveg jellemzőit olvashatjuk ki.

Az éppen kijelölt szöveget így tehetjük dőlt betűssé:

```
RichEdit1.SelAttributes.Style := [fsItalic];
```

***Tipp:** A RichEdit példaprojektet célszerű lehet tanulmányozni, ha a TRichEdit-tel közelebbi ismeretséget akarunk kötni. A projekt maga egy teljesen funkcionális mini-szövegszerkesztőt valósít meg, és csupán néhány száz sort tartalmaz.*

■ *A helyesírás-elemző OCX*

A fejezet zárásaként következzen a beígért példa, amely a Delphihez mellékelt helyesírás-elemző OCX-et használja. A példa egyszerű: egy szerkesztési ablak egyetlen szavát fogjuk elemezni. Természetesen ez nem meríti ki ennek az eszköznek a lehetőségeit. Az sem kíván sokkal nagyobb erőfeszítést, hogy segítségével megvalósítsuk egy memóablak teljes tartalmának ellenőrzését. Egyetlen rosszul írt szót (feltétel, hogy az ablak tényleg csak szót tartalmazzon) ellenőrzése (és kijavítása) összesen két programsor megírását igényli!

A projekt elkészítéséhez

1. helyezzünk egy TEditBox komponenst a formra,
2. ugyanezt tegyük meg az OCX helyesírás-ellenőrző vezérlőelemmel (amelyet a komponenspaletta OCX oldalán találunk meg),
3. adjunk a formhoz egy nyomógombot is.

Ezután adjuk a következő kódot a gomb OnClick eseménykezelőjéhez:

```
procedure TForm1.ButtonClick(Sender: TObject)
begin
  VCSpeller1.CheckText := Edit1.Text;
  If VCSpeller1.ResultCode = 0 then
    Edit1.Text :=VCSpeller1.ReplacementWord;
end;
```

Ez csak egyetlen szó kijavítására alkalmas, ha a szerkesztési ablak ennél hosszabb szöveget tartalmaz, nem működik. Egy jó hír azok számára, akik hosszabb szövegek elemzését is szeretnék megvalósítani: a komponens maga szét tudja választani a szavakat, nem kell ehhez külön kereső programot írnunk. Az elemzés ilyen módon automatikusan a szöveg egyes szavain végigsöpörve történik.

Tipp: *A komponens különálló help fájlja (VSOCX.HLP) a VSpell alkönyvtárban található.*

14. FEJEZET

Adatbázis-kezelési lehetőségek

Ebben a fejezetben sorra veszünk néhányat a Delphi adatbázis-kezelési lehetőségei közül, nagyobb hangsúlyt helyezve az adatkezeléssel kapcsolatos komponensekre. Ezekkel (miután a Delphi rácsatlakozott valamilyen adatbázisra) adatokat tudunk beolvasni és feldolgozni az adatbázisból, és oda újabb adatokat tudunk küldeni. Ez a fejezet nem tekinthető többnek egy kedvcsináló bevezetésnél. Ha komolyabban kívánunk foglalkozni a Delphiben rejlő adatbázis-kezelési lehetőségekkel (bármely változatról is legyen szó), hamar rádöbbenünk, hogy speciális adatbázis programozási ismeretek nélkül ez nem megy. Mélyebb tudásra olyan könyvekből tehetünk szert, amelyek kimondottan ezzel a területtel foglalkoznak. Az adatbázis-programozás – mindegy, hogy a Delphiben végezzük vagy más erre kifejlesztett eszközben – egyáltalán nem triviális. A Delphi két verziójában rejlő adatbázis-programozásilehetőségek leírása legalább kétszer ekkora könyvet igényelne.

Megjegyzés: *Amennyiben a Delphi kliens/szerver változatával dolgozunk, még több lehetőség áll a rendelkezésünkre – ez a verzió nem tárgya a könyvnek.*

Megjegyzés: *A Tool menün keresztül elérhető Database Desktop praktikus eszköz adatbázis-kezelő alkalmazások készítéséhez. Segítségével létrehozhatunk, átalakíthatunk, indexelhetünk, módosíthatunk és lekérdezhetünk adatbázis-táblázatokat. A Database Desktop alkalmas arra is, hogy egyik adatbázist egy más struktúrájú és formátumú adatbázissá alakítson.*

■ *Néhány szó a modern adatbázisokról*

Mielőtt jobban belemélyedünk a Delphi adatbázis-kezelő lehetőségeibe, érdemes körülnézni, hogyan működnek a modern adatbázisok. Ez a rész – anélkül, hogy elvesznénk a technikai részletekben – megpróbálja bemutatni, hogy mit nevezünk relációs adatbázis-modellnek. Ezt használja a Delphi, a Paradox, a Microsoft Access és sok más programozható PC-s adatbázis, mint a dBase és a FoxPro.

Először – mielőtt a kifinomultabb relációs modellt bemutatnánk – nézzünk néhány egyszerűbb modellt. A Rolodex egy jó példája a *flat file* típusú adatbázisoknak. Ezt a típust indexelt kártyák halmazának tekinthetjük. Létrehozására ideálisak a típusos fájlok, ezeket könnyű kezelni és módosítani, valamint erőforrásigényeik sem számottevőek. Ezekben az adatbázisokban az adatok halmazként tárolódnak. Indexek hozzáadásával növelni lehet egyes rekordok hozzáférési sebességét, de használatuk nem lényeges – különösen nem kis adathalmazok esetén.

Az ilyen adatbázisok használatánál a legnagyobb problémát a lehetőségek korlátozottsága okozza. Tegyük fel például, hogy valamilyen vállalkozásunk van, és két nyilvántartási listánk. Az egyik indexelt rendszerben a vásárlókat, a másikban a számlákat tároljuk. Ha valakinek megváltozik a címe, nyilván kívánatos, hogy automatikusan módosítani lehessen mind a két listát, amíg azonban az adatok két külön adatbázis részei, ezt csak nagy erőfeszítés árán valósíthatjuk meg.

Fejlettebb adatbázisok – amilyeneket például a Database Desktopban (vagy a Paradoxban vagy magában a Delphiben) készíthetünk – nem illenek bele az indexelt kártya modellbe. Ezekkel egyszerűen megoldható az előbb említett aktualizálási probléma. Sok más előnnyel is büszkélkedhetnek, habár a nagyobb rugalmasság általában nagyobb hardver- és kódigényt is jelent egyben.

Nem egyszerű azon adatbázisok struktúrájának jellemzése, amelyeket a Delphihez mellékelt Borland Database Engine (BDE) segítségével (amely az olvasó winchesterén is megtalálható) lehet létrehozni. Illusztrálásként gondoljunk most úgy az adatbázisra, mint egy nagyméretű alaktalan adathalmazra, amely ott van egy feneketlen ládában valmilyen elkent köd formájában. A bonyolult zűrzavar felett azonban egy nagyhatalmú varázsló őrködik, aki a káoszból is rendet tud teremteni.

Az adatbázisban sokféle adat található, az összes kiadott könyvvel, köztük az összes számítástechnikai kiadvánnyal kapcsolatos adatok is ott vannak valahol a láda mélyén. Egy nap megjelenünk nála azzal a szándékkal, hogy információkat gyűjtsünk a számítástechnikai könyvekről. A rengeteg könyv között sok olyan található, amelyek címe és témája megegyezik. Szerzőkből sincs kevés. Az óriási adathalmaz nagyon sokféle kérdésre tartalmazza a választ, amit azonban nem könnyű megtalálni. A varázsló nagy erejének és végtelen adattárolójának köszönhetően minden információval rendelkezik a számítógépes könyvekről. Ismeri a szerzőket, címeket, oldalszámokat, kiadókat és bármit, amit csak el lehet képzelni. A varázsló a könyvek bármely részét érintő kérdésre tudja a választ. Rákérdezhetük egy szerző összes könyvére. Felsoroltathatjuk vele az összes olyan könyvet, amelynek címében egy megadott betűsorozat található, vagy amelyet egy bizonyos kiadó publikált. Kérhetjük, hogy adja meg azon művek listáját, amely az előző három feltételt egyszerre kielégíti.

Képzeljük el: az a kívánságunk, hogy sorolja fel az összes olyan könyvet, amelyet az Osborne/McGrawHill adott ki 1995-ben, adja meg a szerzőt, a címet és az oldalszámot is. A varázsló végignézi a láda tartalmát, majd egy táblázatszerű elrendezésben megadja a választ. Nem is értjük és nem is érdekel minket, hogyan tárolta és dolgozta fel ezeket az adatokat. A lényeg a végeredmény: egy könnyen kezelhető táblázat.

Vegyük észre, hogy egy típusos állomány nem képes egy ilyen jellegű szituáció kezelésére. Ha minden szerzőhöz egyetlen rekord tartozik, akkor nincs módunk megtudni, hogy hány mezőt hagyjunk művei nyilvántartása számára, hiszen elképzelhető, hogy folyamatosan ontja magából az új könyveket. Természetesen megtehetnénk, hogy könyvek szerinti rekordokat hozunk létre, de ekkor sok ismétlődés lenne – a szerző fontosabb adatait minden könyvének rekordjában meg kellene jeleníteni. Ha az adatokat valamilyen alaktalan masszaként tárolnánk, a varázslónk, aki ügyes könyvelési trükköket használna a kereséshez, a fölösleges ismétlések elkerülésére, az adatok összesűrítésére stb.

Megjegyzés: A modern adatbázis-terminológiában a kérdést lekérdezésnek (queries), az eredményt adattáblának vagy adattáblázatnak (table, view) nevezik. A varázsló különböző lekérdezésekre különböző adattáblákkal válaszolhat, habár a ládában csak egy (nagyméretű) adatbázis van.

A relációs adatbázisok lekérdezésére általánosan használt az ún. SQL nyelv. Az SQL az angol „structured query language"-ből (=strukturált lekérdező nyelv) alkotott betűszó. Lényegében ezt a nyelvet építették a Delphibe is.

A valóságban természetesen nincsenek varázslók, és az adatok sem ilyen megfoghatalan köd formájában léteznek. Miként azt a következő fejezetrészben láthatjuk, az adatok valójában egymástól nem független táblázatok formájában tárolódnak, amelyeket az adatbázis-kezelő kapcsol össze. A táblázatok oszlopait mezőknek (fields), míg sorait rekordoknak nevezzük.

A Delphi adatbázis architektúrája

A Delphi sokuldalúsága abban is megmutatkozik, hogy sokféle adatbázisra rácsatlakozhat. Formokon és a programozó által készített jelentéseken keresztül, a BDE hozzáférhet olyan lokális adatbázisokhoz mint a Paradox és DBase, hálózati SQL szerver adatbázisokhoz mint az InterBase, a Sysbase és minden ODBC-n (Open Database Connectivity) keresztül elérhető adatbázishoz is.

Adatbázis-alkalmazások készítéséhez – ugyanúgy, mint programok előállításához – komponenseket használunk. Az adatbázis-komponenseknek is vannak tulajdonságaik, amelyeket tervezési vagy futási időben állíthatunk be illetve módosíthatunk. A Delphi komponenspalettáján két külön oldal található az adatbázis-eszközök számára. A Data Access oldalon rendelkezésre álló komponensekkel az adatbázisokhoz történő csatlakozást valósíthatjuk meg, míg a Data Controls oldal vezérlőelemeit a kész információk megjelenítésére használhatjuk. A két oldalon lévő komponenseket közös néven adatbázis-komponenseknek fogjuk hívni.

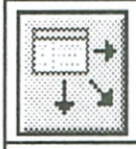
A Data Access komponensek

A Data Access oldalon lévő komponensek segítségével adatbázisokhoz csatlakozhatunk. Ezek a komponensek megkönnyítik számunkra a hozzáférést az adatbázis táblázataihoz és a táblázatok rekordjaihoz. A Data

Access komponensek nem vizuálisak – azaz futás közben nem jelennek meg a képernyőn.

A konkrét adatbáziskezelő alkalmazás készítésének első lépése, a Data Access komponensek formra helyezése, és annak tulajdonságainak beállítása a használni kívánt adatbázisnak, táblázatnak és rekordoknak megfelelően. A 14-1. táblázat ezeket a komponenseket és alkalmazási területeiket sorolja fel. A TSession komponenstől eltekintve, a két adatbázis panel a Delphi minkét változatában ugyanazokat az elemeket tartalmazza.

14-1. táblázat. A Data Access komponensek

Ikon	*Komponens*	*Leírás*
	TDataSource	A TTable, TQery, TStoredProc komponenseket köti össze a vizuális komponensekkel.
	TTable	Adatokat olvas ki egy táblázatból, és azokat a TDataSource komponenseken keresztül a vizuális a adatbázis komponensekhez továbbítja.
	TQuery	SQL utasításokat használ az adatok táblázatból történő kiolvasására, és azokat a TDataSource komponenseken keresztül a vizuális a adatbázis komponensekhez továbbítja.
	TStoredProc	Hozzáférést bitosít a szerver eljárásaihoz. Adatokat küld az adatbáziba a vizuális komponensekből.
	TDataBase	Elsősorban kihelyezett adatbázokkal való kapcsolat teremtéshez használjuk.
	TSession	Nem vizuális komponens. Csak a Delphi 32-ben elérhető. Globális felügyeletet biztosít az adatbázis-kapcsolatok felett.
	TBatchMove	Adattáblákat struktúrákat mozgat és másol egyik adatbázisból a másikba.
	TUpDateSQL	Ezen keresztül frissíthetjük fel az SQL lekérdezéseket.
	TReport	Jelentések nyomtatását és megjelenítését végzi a ReportSmith-en keresztül.

A TTable és a TQuery a Data Access oldal leggyakrabban használt komponensei az adatbázishoz történő csatlakozás realizálására. A TTable és a TQuery komponens az adatokat közvetíti számunkra a BDE-hez való közvetlen kapcsolódással. A TDataSource komponens egy hidat képez a vizuális komponensek és az előzőek között. A következőkben ezzel a három komponenssel ismerkedhetünk meg közelebbről.

A TTable komponens

Az TTable komponens segítségével egyszerűen létesíthetünk kapcsolatot egy adatállománnyal. A TTable használatához a következőket kell tennünk:

1. Helyezzünk egy TTable komponenst a formra.
2. A DatabaseName tulajdonságban adjuk meg az adatbázist tartalmazó könyvtár nevét.
3. Állítsuk a TableName tulajdonság értékét az adatbázis kívánt táblájának nevére.
4. Kattintsunk kettőt a TTable komponensre a formon, ezáltal a táblázat kimeneti formátumát állíthatjuk be az adatkezelő komponenseknek megfelelően.
5. Az Active tulajdonság True-ra állításával, nyissuk meg a táblázatot.

(A TTAble gyorsmenüjét használva különböző mezőket adhatunk az adattáblához.)

Megjegyzés: *Ha a formon kettőt kattintunk egyTTable, TQuery, vagy egy TStoredProc komponensre, megjelenik a Fields Editor ablak. Ez eltér attól a gyakorlattól, hogy a komponensre történő kattintás a kódszerkesztő ablakot nyitja meg az alapértelmezett eseménykezelő sablonnal. A DataAccess komponensek közül egyedül a TDataSource komponens viselkedik így.*

A TQuery komponens

A TQuery komponens lehetővé teszi, hogy SQL utasításokat adjunk ki az SQL szerver számára, mint például a SELECT utasítást, amely az adatállomány egy kívánt részét adja vissza. A komponens használata:

1. Helyezzük a TQuery komponenst a formra.
2. A DatabaseName tulajdonságban adjuk meg az adatbázist tartalmazó könyvtár nevét.

3. Az SQL tulajdonságban adjuk meg a kívánt SQL utasításokat. Kattintsunk a listagombra a String Editor dialógusablak megnyitásához. Például: 'SELECT*FROM CUSTOMER.DB'
4. Kattintsunk kettőt a TQuery komponensre a formon, hogy a táblázat kimenetének formátumát beállíthassuk az adatkezelő komponenseknek megfelelően.
5. Az Active tulajdonság True-ra állításával nyissuk meg a lekérdezést.

A TDataSource komponens

A TDataSource komponens adatokat közvetít a TTable, TQuery és TStoredProc komponensek és a vizuális adatbázis-alkotóelemek között. Használata:

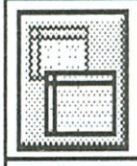
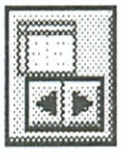
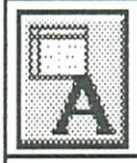
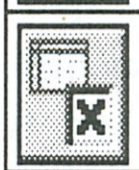
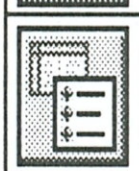
1. Helyezzük a TDataSource komponenst a formra.
2. Adjuk meg DataSet tulajdonságban a megfelelő TTable, TQuery vagy TStoredProc komponens nevét. A DataSet tulajdonság lenyíló listaablakában fel vannak sorolva a formon lévő DataAccess komponensek, amelyek közül választhatunk.

A Data Controls oldal komponensei

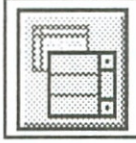
A Data Control oldal komponensei az adatok megjelenítését, szerkesztését és az adatbázisba történő felvétel elindítását valósítják meg. Egyes Data Control vezérlő elemek valójában közönséges komponensek adatbázis-kezeléshez igazított változatai. Ezek mind láthatóak.

A Delphi adatbázisprogramok egy fontos részét a TTable vagy TQuery komponensek által szolgáltatott adatok megjelenítése képezi. A DDE (Delphi Database Engine) gondoskodik az adatok eljuttatásáról az adatkezelő vezérlőelemekhez a TDataSource komponensen keresztül. A Data Controls komponensek teremtik meg a vizuális kapcsolatot a felhasználó és az adatállomány között. A 14-2. táblázat ezen komponensek közül azokat mutatja be röviden, amelyek mindkét Delphi változatban megtalálhatók. A 14-3. táblázat kizárólag a Delphi 32 Data Controls komponenseit foglalja össze, a 14-4. táblázat pedig csak a Delphi 16-ban elérhetőeket sorolja fel.

14-2. táblázat. A mindkét Delphi változatban megtalálható Data Controls komponensek

Ikon	*Komponens*	*Leírás*
	TDBGrid	Táblázat formájában jeleníti meg az adatokat, amelyeket szerkeszthetünk is a komponensben.
	TDBNavigator	Egy kapcsolótábla megjelenésű komponens, amely segítségével egy adattáblázatban mozoghatunk rekordmutatót léptethetjük előre vagy visszafelé, szerkeszthetünk s beilleszthetünk rekordokat.
	TDBText	Az adatállomány egy mezőjének statikus megjelenítésére szolgál.
	TDBEdit	Az aktuális rekord adatainak szerkesztését teszi lehetővé.
	TDBMemo	Memó komponens, amelyben az aktuális rekordból származó BLOB (binary large object=nagy bináris objektum) szerkesztésére alkalmas. (A BLOB gyakran valamilyen grafikát jelent.)
	TDBImage	Olyan TImage komponens, amely az aktuális rekord bittérkép formátumú BLOB-jának megjelenítésére és szerkesztésére alkalmas.
	TDBListBox	Olyan listaablak-komponens, amely egy adattábla oszlopának adatait jeleníti meg.
	TDBComboBox	Olyan TComboBox komponens, amelyben egy adattábla oszlopának adatait jeleníthetjük meg és szerkeszthetjük.
	TDBCheckBox	Olyan jelölő négyzet komponens, amelyben az aktuális rekord egy logikai mezőjét jeleníthetjük meg és állíthatjuk be.
	TDBRadioGroup	TRadioGroupBox komponens, amelyek segítségével egy adattáblázat oszlopának adatait jeleníthetjük meg és állíthatjuk be.

14-3. táblázat. Data Controls komponensek a Delphi 32 bites változatában

Ikon	*Komponens*	*Leírás*
	TDBLookupListBox	Olyan listaablak, amellyel egy másik adattábla mezőire hivatkozhatunk.
	TDBLookupComboBox	Olyan csatolt listaablak, amellyel egy másik adattábla mezőire hivatkozhatunk.
	TDBCtrlGrid	Több rekordot megjelenítő táblázatkomponens.

14-4. táblázat. Data Controls komponensek a Delphi 16 bites változatában

Ikon	*Komponens*	*Leírás*
	TDBLookupList	Olyan listaablak, amellyel egy másik adattábla mezőire hivatkozhatunk.
	TDBLookupCombo	Olyan csatolt listaablak, amellyel egy másik adattábla mezőire hivatkozhatunk.

■ *Database Form Expert*

A Database Form Expert (elérhető a Delphi 32 Database menüjében illetve a Delphi 16 Help menüjében) leegyszerűsíti a táblázatalapú vagy egyéb adatkezelő formok létrehozását, amelyek egy már meglévő adatbázist használnak. Tapasztalt és tapasztalatlan felhasználók számára egyaránt előnyös lehet a Database Form Expert (a későbbiekben csak DFE) használata. A DFE-vel egyszerű formokat vagy „master/detail" formokat hozhatunk létre, amelyek egy ritkán változó törzstáblából (master table), vagy egy gyakran változó adattáblából (detail table) vonják ki az adatokat a TTable és a TQuery Data Access komponensek használatával.

DFE-t a megfelelő menü Database Form Expert pontjának választásával kezdhetünk. Ezután a kérdésekre adott választól függően:

- Készíthetünk „master" és „detail" formokat (definíciót l. feljebb).
- TTable vagy TQuery komponensből meríthetünk adatokat.
- Meghatározhatjuk, hol van az adatállomány.

- Kijelölhetjük a mezőket, és megadhatjuk az adatok megjelenési formátumát.

A Database Form Expert automatikusan elhelyezi a szükséges komponenseket a formon, és összekapcsolja a TDataSource komponenst (komponenseket) a Data Access komponensekkel (TTable, TQuery, TStoredProc) és a vizuális vezérlőelemekkel, valamint a TabOrder (fókuszátadási sorrend) tulajdonságokat is beállítja.

■ *Végső megjegyzés*

Ez a fejezet a Delphi adatbázis-kezelő eszközeinek lehetőségeiből csak egy rövid ízelítőt adott. A Delphi adatbázis-kezelő komponensei sok feladat automatizálását teszik lehetővé, amit egyébként csak hosszú programozás árán érhetnénk el. A legegyszerűbb adatbázis alkalmazások még kódírást sem igényelnek. A bonyolultabbak megírásánál sem valószínű, hogy csalódni fogunk a Delphi erejében.

Tárgymutató

.DFM kiterjesztés 46, 72
.DPR kiterjesztés 49, 73
.PAS kiterjesztés 47, 72

A

Abort
  eljárás 305
  metódus (TPrinter objektum) 193
ActiveControl tulajdonság
  TScreen objektum 189
  Form 95
ActiveForm tulajdonság (Tscreen objektum) 189
Adatbázis kezelés 410
Adattáblázat 413
Add Breakpoint (hibakeresés) 308
Add File menüpont (File menü) 77
Add File To Project ikon (eszközpaletta) 44, 69
Add metódus (sztringlisták) 222
Add To Project menüpont (File menü) 52
Add To Repository menüpont (Project menü) 59
Add Watch (hibakeresés) 308
Additional oldal (komponenspaletta) 105
Addr függvény 228
Alakzatok rajzolása 377
Alaposztály 269
Align menüpont (Edit menü) 55, 79
Align to Grid menüpont (Edit menü) 55, 79
Align tulajdonság 115
  TPanel komponens 382
Aligned Record Fields fordítási direktíva 260
Alignment Palette
  Delphi 16 73
  Delphi 32 49
  menüpont (View menü) 57, 82
Alignment tulajdonság (TLabel komponens) 117
Állapotjelző sor 382, 384
Általános dialógusablakok 385
AnsiChar típus 153
AnsiString 154
API 240
Append művelet (fájlkezelés) 333
Argumentum 236
Assign metódus (clipboard) 348
Assignable Typed Constants fordítási direktíva 260
AssignFile eljárás (fájlkezelés) 322
AsText metódus (clipboard) 347
Automatikus mozgatás (egérműveletek) 401
AutoScroll tulajdonság 96
AutoSelect tulajdonság (TEdit komponens) 118
AutoSize tulajdonság
  TImage komponens 375
  TLabel komponens 117

B

BDE 411
Beágyazás (OLE) 360
Beágyazott menük 134

Beágyazott tulajdonságszerkesztő 92
BeginDrag metódus
(egérműveletek) 400, 401
Bit-műveletek 171
Boolean típus 148
BorderIcons tulajdonság 96
BorderStyle tulajdonság 96
Borland Database Engine 411
break kulcsszó 262
Break tulajdonság (menük) 133
Breakpoints menüpont (View
menü) 57, 82
Brief 54, 62, 85
Bring To Front menüpont (Edit
menü) 55, 79
BringToFront metódus 186
Browse Gallery 201
Browse Symbol menüpont (Search
menü) 56, 81
Browser menüpont (View menü) 57, 82
Build All menüpont (Project menü) 59
Byte típus 149

C

Call Stack
funkció (hibakeresés) 307
menüpont (View menü) 58, 82
Cancel tulajdonság (TButton, TBitButton
komponens) 121
Canvas objektum 375
Caption tulajdonság 28, 96, 116
case utasítás 183, 184
Center tulajdonság (TImage
komponens) 375
ChangeFileExt függvény (fájlkezelés) 324
Char típus 151
CharCase tulajdonság (TEdit
komponens) 118
ChDir (fájlkezelés) 322
Checked tulajdonság (menük) 133
Ciklus 174
determinisztikus 174
elöltesztelő 174
hátultesztelő 174
nem determinisztikus 174, 178
Clear metódus (sztringlisták) 223
ClientHeight tulajdonság 96
ClientWidth tulajdonság 96
Clipboard 346
formátumok 348
Close all menüpont (File menü) 52
Close File menüpont (File menü) 77
Close menüpont (File menü) 52
Close metódus 208
Close Project menüpont (File menü) 76
CloseLink metódus (DDEClientConv) 356
ColCount tulajdonság (TStringGrid,
TDrawGrid komponens) 127
Color tulajdonság 93, 96, 115
grafika 378
TColorDialog komponens 389
ColorToRGB függvény (grafika) 376
Columns tulajdonság (TListBox,
TComboBox komponens) 123
Comp típus 151
Compile menüpont (Project menü) 59
Complete Boolean Eval fordítási
direktíva 259
Component List menüpont (View
menü) 58, 82
Component menü 42, 60
Component Palette
Delphi 16 70
menüpont (View menü) 59, 83
tulajdonság (TScreen objektum) 189
Component Writers Guide 257
ComponentCount tulajdonság (TScreen
objektum) 189
ComponentIndex tulajdonság (TScreen
objektum) 189
ConnectMode tulajdonság
DDEServerConv 355
TDDEClient 350
const
blokk 164
kulcsszó 161
constructor kulcsszó 274
continue kulcsszó 183
Copies tulajdonság (TPrint
komponens) 390

Copy menüpont (Edit menü) 54, 79
CopyToClipboard metódus (DDEServerItem) 354
Count tulajdonság (sztringlisták) 221
Create metódus 186, 288
Creation Order menüpont (Edit menü) 55, 80
Csatolás (OLE) 359
Csendes kivételek 305
Csonkprogramozás (hibakeresés) 309
Ctl3D tulajdonság 116
Currency típus 154
Cursor tulajdonság 96, 115
  TScreen objektum 189
Cursors tulajdonság (TScreen objektum) 190
Cut menüpont (Edit menü) 54, 79

D

Data Access komponensek (adatbázis kezelés) 413
Data Controls komponensek (adatbázis kezelés) 413, 416
Database Desktop 42, 411
Database Form Expert 42, 418
Database menü Delphi 32 42
DDE 349
  elem 350
  kapcsolat fejlesztési időben 350, 351
  kapcsolat futási időben 352
  komponensek eseményei 353
  komponensek metódusai 353
  komponensek tulajdonságai 353
  témája 350
DDEClientItem komponens 351
DDEConv tulajdonság (DDEClientItem) 357
DDEItem tulajdonság (DDEClientItem) 358
DDEService tulajdonság (DDEServerConv) 350, 355
DDETopic tulajdonság (DDEServerConv) 355
Debugging Information fordítási direktíva 259
Default tulajdonság (TButton, TBitButton komponens) 121
DefaultColWidth tulajdonság (TStringGrid, TDrawGrid komponens) 127
DefaultExt tulajdonság (TOpenDialog, TSaveDialog komponens) 388
DefaultRowHeight tulajdonság (TStringGrid, TDrawGrid komponens) 127
DefaultText tulajdonság (TRichEdit komponens) 407
Delete menüpont (Edit menü) 55, 79
Delete metódus (sztringlisták) 223
Delphi 1.0 15, 39
Delphi 2.0 15, 64
Delphi fordító 256
Dialogs oldal (komponenspaletta) 105
Dialógusablak 91
Dinamikus adatkapcsolatok 349
DirectoryFileListBox komponens (fájlkezelés) 330
DiskFree függvény (fájlkezelés) 323
DiskSize függvény (fájlkezelés) 324
DLL 248
  külső DLL-ek használata 248
Double típus 151
downto kulcsszó 176
DragCursor tulajdonság (egérműveletek) 400
DragMode tulajdonság (egérműveletek) 400
DriveComboBox komponens (fájlkezelés) 330

E

EAbort kivétel 305
EBreakPoint kivétel 301
Ecset (grafika) 379
Edit menü
  Delphi 16 78
  Delphi 32 42, 50
EDivByZero kivétel 299
EFault kivétel 300
Egérműveletek 393
Egész aritmetika kivételek (kivételhierarchia) 298
Egész típusok 149

EGPFAult kivétel 300
Egyszerű szerkesztő 90
EIntOverFlow kivétel 299
EInvalidOp kivétel 299
EInvalidOpCode kivétel 301
EInvalidPointer kivétel 298
Ellipse metódus (Canvas objektum) 377
Elrendezéskezelő paletta (Delphi 16) 73
else kulcsszó (hibakezelés) 295
Enabled tulajdonság 97, 115
  TTimer komponens 125
EndDrag metódus (egérműveletek) 400
Environment menüpont (Options menü) 84
Environment Options dialógusablak 52, 61, 84
Eof függvény (fájlkezelés) 337
EOutOfMemory kivétel 298
EOverflow kivétel 299
EPageFault kivétel 300
Epsilon 54, 62, 85
ERangeError kivétel 299
Erease (fájlkezelés) 322
Értékadás 158
Értékadáskompatibilitás 158
Eseménykezelő eljárás 29, 161
Eseményzuhatag 317
ESingleStep kivétel 301
EStackFault kivétel 300
Eszközpaletta
  Delphi 16 68
  Delphi 32 43
  létrehozása 382
EUnderflow kivétel 299
Evaluate/Modify
  hibakeresés 308
  dialógusablak 311, 312
except kulcsszó 294
ExecuteMacro metódus (DDEClientConv) 357
ExecuteMacroLines metódus (DDEClientConv) 357
ExeName tulajdonság (TApplication objektum) 188
Exit menüpont (File menü) 52, 77
ExpandFileName függvény (fájlkezelés) 324
exports kulcsszó 257
Extended Syntax fordítási direktíva 234, 259
Extended típus 151
ExtendedSelect tulajdonság (TListBox, TComboBox komponens) 123
ExtractFileExt függvény (fájlkezelés) 324
ExtractFileName függvény (fájlkezelés) 324
ExtractFilePath függvény (fájlkezelés) 324
EZeroDivide kivétel 299

F

Fájlkezelés 320
  parancsok 321
Fájlrendszer komponensek (fájlkezelés) 327, 328
  összekötése 331
Fejléc (Delphi 16) 66
Fejléc 41
Feltételes direktívák 262
Feltételes fordítás 261
Feltételes utasításvégrehajtás 180
Felügyelőablak 290, 291
Felüldefiniálás 270
File menü (Delphi 16) 75
File Open dialógusablak 387
File Save dialógusablak 387
FileAge függvény (fájlkezelés) 324
FileDateToDateTime függvény (fájlkezelés) 324
FileExists függvény (fájlkezelés) 325
FileGetAttr függvény (fájlkezelés) 325
FileGetDate függvény (fájlkezelés) 324
FileListBox komponens (fájlkezelés) 328
FileSearch függvény (fájlkezelés) 325
FileSetAttr függvény (fájlkezelés) 325
Filter tulajdonság
  OpenDialog komponens 35
  TOpenDialog, TSaveDialog komponens 388
FilterComboBox komponens (fájlkezelés) 330
FilterIndex tulajdonság (TOpenDialog, TSaveDialog komponens) 388
FilterName tulajdonság (TOpenDialog, TSaveDialog komponens) 388
finally kulcsszó 232, 233, 294

Find Error menüpont (Search menü) 56
Find Error menüpont (Search menü) 81
Find menüpont (Search menü) 56
Find menüpont (Search menü) 80
FindFirst függvény (fájlkezelés) 326
FindNext függvény (fájlkezelés) 326
FixedColor tulajdonság (TStringGrid, TDrawGrid komponens) 127
FixedCols tulajdonság (TStringGrid, TDrawGrid komponens) 127
FixedRows tulajdonság (TStringGrid, TDrawGrid komponens) 127
FloatToStr függvény (típuskonverziók) 174
Focused metódus 186
főform 205
Fogd és vidd műveletek 398
Font tulajdonság 91
Font tulajdonság 97
Font.Color (TFontDialog dialógusablak) 389
Font.Name (TFontDialog dialógusablak) 389
Font.Size (TFontDialog dialógusablak) 389
Fonts menüpont (View menü) 58
Fonts tulajdonság (TScreen objektum) 190
for ciklus 174, 175
Force Far Calls fordítási direktíva (Delphi 16) 260
Fordítási direktívák 258
  hatásuk az objektumböngészőre 291
Form 25, 95, 136
  futási időben 206
  mentése 203
  modális 208
  sablonok 204
  tervezése 88
  törlése 204
  tulajdonságai 95
FormatChars tulajdonság (DDEServerConv) 355
Formdeklaráció 281
Formesemények 101
Forms
  menüpont (View menü) 83
  tulajdonság (TScreen objektum) 189
FormsCount tulajdonság (TScreen objektum) 189
FormStyle tulajdonság 97
forward kulcsszó 251
FreeMem eljárás (memória-felszabadítás) 232, 233
FromPage tulajdonság (TPrint komponens) 390
  Futás idejű könyvtár 238
  rutinok 238

G, Gy

GetDir eljárás (fájlkezelés) 323
GetMem eljárás (memória-foglalás) 232, 233
GetSystemInfo API függvény 241
GetSystemMetrics API függvény 241
Glyph tulajdonság (TBitButton komponens) 122
Go to Line Number menüpont (Search menü) 56, 81
goto utasítás 262
Grafika 371
GridLineWidth tulajdonság (TStringGrid, TDrawGrid komponens) 128
Gyermekform (MDI) 390
Gyermekkomponens 110
Gyorsítóbillentyűk 132

H

Halmaztípus 215
  műveletei 215
Handle tulajdonság (grafika) 378
Hardver kivételek (kivételhierarchia) 300
HasFormat metódus (clipboard) 348
Heap kivételek (kivételhierarchia) 298
Height tulajdonság 97
  TScreen objektum 190
  TStringGrid, TDrawGrid komponens 127
Help Compiler 404
Help menü 42
HelpCommand metódus (TApplication objektum) 406

HelpContext metódus (TApplication objektum) 406
HelpContext tulajdonság 97
HelpFile tulajdonság (TApplication objektum) 188
HelpJump metódus (TApplication objektum) 406
Helyben történő aktivizálás (OLE) 359
Hibajavítási technikák 306
Hibakeresés 307
Hibakezelés 293
Hide metódus 186
Hide metódus 208
HideSelection tulajdonság (TEdit komponens) 118
History List menüpont (File menü) 77
HorzScrollBar tulajdonság 98
Hosszú fájlnév 320
Huge Strings fordítási direktíva 260

I

I/O Checking fordítási direktíva 258
I/O ellenörzés (fájlkezelés) 336
I/O függvények (fájlkezelés) 323
I/O kivételek (kivételhierarchia) 298
Icon tulajdonság (TApplication objektum) 188
Icon tulajdonság 98
IDE Classic 54, 62, 85
if-then utasításpár 181, 182
Image Editor 85
Image komponens 35
implementációs blokk 163
Incremental Search menüpont (Search menü) 56
Incremental Search menüpont (Search menü) 81
IndexOf függvény (sztringlisták) 222
Information menüpont (Project menü) 60
InitialDir tulajdonság (TOpenDialog, TSaveDialog komponens) 388
Initialization blokk 256
Inputablak 194
InputBox függvény 198
Inputfókusz 116
InputQuery függvény 199
Insert metódus (sztringlisták) 223
Insert Object dialógusablak (OLE) 362, 363
Install menüpont (Component menü) 60
Integer típus 149
interface blokk 164
Interface blokk 245
Interface blokk 255
Interval tulajdonság (TTimer komponens) 125
Intervallumtípus 214
IntToHex függvény (típuskonverziók) 174
IntToStr függvény (típuskonverziók) 174

K

Karakterláncok 167
Képek elmentése (grafika) 380
Késői kapcsolatteremtés 271
KeyPreview tulajdonság 98
Kind tulajdonság (TBitButton komponens) 122
Kivétel
  objektum 293
  felhasználó által definiált 304
  előidézése 301
Kivételhierarchia 297
Kivételkezelés 293
Kivételobjektumok deklarálása 304
Kivételosztályok kezelése 301
Kliensterület 41
Kódszerkesztő ablak 30, 47
Komponensek 103,104
  elrendezése 114
  manipulálása 111, 112, 113
Komponenspaletta 33, 45 (Delphi 32)
Konstansok 159
  típusos 165
Konstruktorok 273
Konténerkomponens 110
Konverziós kivételek (kivételhierarchia) 300
Külsőaktivizálás (OLE) 358

L

Late binding 271
Layout tulajdonság (TBitButton komponens) 122

Lebegőpontos matematikai kivételek (kivételhierarchia) 299
Lebegőpontos típusok 150
Left tulajdonság 99, 115
Legördülő lista 91
Legutoljára megváltoztatott pont (grafika) 378
Lekérdezés (adatbázis kezelés) 413
library kulcsszó 257
Lines tulajdonság
DDEClientItem 358
DDEServerItem 354
TMemo komponens 119
Lock Controls menüpont (Edit menü) 56, 80
Logikai
hibák 309
operátorok 170
relációk 178
tulajdonságok 146
LongInt típus 149
LPR (grafika) 378

M

Manuális mozgatás (egérműveletek) 401
Margin tulajdonság (TBitButton komponens) 122
MaxLength tulajdonság (TEdit komponens) 118
MaxPage tulajdonság (TPrint komponens) 390
MDI formok 390
Megjegyzések 142
Memóriafelszabadítás 232
Memóriafoglalás 232
Menü 128
Menu Designer 129, 131
Menü tulajdonság 99
Menüpont 133
Menürevező 129, 131
Menüsablonok 135
MessageDlg föggvény 194
MessageDlgPos függvény 197
Metódus 37
Metódusok 185
Minimize metódus (TApplication objektum) 188
MinPage tulajdonság (TPrint komponens) 390
MkDir (fájlkezelés) 322
Mode tulajdonság (grafika) 378
Move metódus (sztringlisták) 223
MultiSelect tulajdonság (TListBox, TComboBox komponens) 123
Mutató típusok 152
Mutatók 227
Műveleti sorrend 172

N

Name tulajdonság 99, 115
New Appplication menüpont (File menü) 50
New Component menüpont (File menü) 76
New Data Module menüpont (File menü) 51
New Edit Window menüpont (View menü) 59, 83
New Form ikon (Eszközpaletta) 45, 69
New Form menüpont (File menü) 51, 76
New menüpont
Component menü 60
File menü 50
New Project menüpont (File menü) 75
New Unit menüpont (File menü) 76
NewPage metódus (TPrinter objektum) 193
Nil érték (mutatók) 228
Nulla-végű sztringek 233
NumGlyph tulajdonság (TBitButton komponens) 122
Nyomkövetési információk 310

O

Object Browser 270, 289
Object Inspector menüpont (View menü) 57, 81
Object menüpont (Edit menü) 56, 80
Objektumböngésző 289
Objektumfelügyelő 24, 46, 89, 90
ablaka (Delphi 16) 70
Objektumkonverziós kivételek (kivételhierarchia) 300
Objektumok 265
csatolása és beágyazása 345, 358
futási időben 185

létrehozása 272
létrehozása futási idő alatt 286, 287
Objektumorientált programozás 265
Objektumváltozók 283
OLE 1.0 358
OLE 2.0 359
használata fejlesztési időben 362
OLE 345, 358
Automation 367
dokumentumok 361
elemek 361
használata 361
kliens 361
objektum 359
osztályok 361
szerver 360
OleAuto unit
on..do utasítások (védett blokk) 294, 296, 301
OnActivate esemény 37, 101
OnChange esemény
DDEServerItem 354
DDEClientItem 358
TEdit komponens 119
OnClick esemény 33, 102, 115
OnClose esemény 102
DDEClientConv 356
DDEServerConv 355
Form 209
OnCloseQuery esemény 102, 209
OnCreate esemény 102
OnDblClick esemény 102, 115
OnDeactivate esemény 102
OnDestroy esemény 102
OnDragDrop esemény (egérműveletek) 400, 102
OnDragOver esemény (egérműveletek) 102, 400, 402
OnEndDrag esemény (egérműveletek) 400
OnEnter esemény 116
OnExecuteMacro esemény (DDEServerConv) 355
OnExit esemény 116
OnHide esemény 103
OnKeyDown esemény 103
OnKeyPress esemény 103
OnKeyUp esemény 103
OnMouseDown egéresemény 102, 394, 395
OnMouseMove egéresemény 102, 394, 397
OnMouseUp egéresemény 102, 394, 395
OnOpen esemény
DDEClientConv 356
DDEServerConv 355
OnPaint esemény 103
OnPokeData esemény (DDEServerItem) 354
OnResize esemény 103
OnShow esemény 103
OnTimer esemény (TTimer komponens) 126
OOP 265
terminológiája 268
Open File ikon (eszközpaletta) 45, 69
Open File menüpont (File menü) 76
Open Library menüpont (Component menü) 60
Open menüpont (File menü) 52
Open Parameters fordítási direktíva 259
Open Project ikon (eszközpaletta) 44, 69
Open Project menüpont (File menü) 75
OpenDialog komponens 35
OpenLink metódus
DDE 350
DDEClientConv 357
Optimalization fordítási direktíva 259
Options
menü Delphi 16 83
menü Delphi 32 42
TFontDialog dialógusablak 389
menüpont (Project menü) 60
menüpont (Tools menü) 61
tulajdonság (TColorDialog komponens) 389
tulajdonság (TOpenDialog, TSaveDialog komponens) 388
tulajdonság (TStringGrid, TDrawGrid komponens) 128
Orientation tulajdonság (TPrinter objektum) 193

Osztályhierarchia 270
Osztályok 268
Overflow Checking fordítási direktíva 258

Ö, Ő

Öröklődés 269
Ősosztály 269

P

PageHeight tulajdonság (TPrinter objektum) 193
PageNumber tulajdonság (TPrinter objektum) 193
PageWidth tulajdonság (TPrinter objektum) 193
PAnsiChar típus 153
Paraméter 195, 236
  const 195, 236, 237
  value 238
  var 195, 236, 237
Parameters (hibakeresés) 308
Parancsbillentyűk 132
Parancsok 141
Parancssor-paraméterek 343
Parent tulajdonság 288
PasswordChar tulajdonság (TEdit komponens) 118
Paste menüpont (Edit menü) 54, 79
Paste Special dialógusablak (OLE) 364
Pause ikon (eszközpaletta) 44, 69
PChar típus 152, 153, 234
Pen tulajdonság (Canvas objektum) 378
Pentium-Safe FDIV fordítási direktíva 258
Picture tulajdonság (TImage komponens) 375
Pixel 96
Pixelgrafika 376
Pixels tulajdonság (Canvas objektum) 376
PixelsPerInch tulajdonság 100
  TScreen objektum 190
PlainText tulajdonság (TRichEdit komponens) 407
Pointer típus 152
PokeData metódus (DDEClientConv) 357
PokeDataLines metódus (DDEClientConv) 357
Polimorfizmus 270
PopUpMenu tulajdonság 100
Position tulajdonság 100
Print menüpont (File menü) 52, 77
Printers tulajdonság (TPrinter objektum) 193
PrintIndex tulajdonság (TPrinter objektum) 193
Printing tulajdonság (TPrinter objektum) 193
PrintScale tulajdonság 100
  TPrinter objektum 193
private kulcsszó 276, 281
ProcessMessages metódus (TApplication objektum) 188
program kulcsszó 257
Program Pause (hibakeresés) 308
Program Reset
  hibakeresés 308
  menüpont 38
Programok betöltése és futtatása 62
Project fájl 49
Project Manager ablak
  Delphi 16 72
  Delphi 32 48,49
Project Manager menüpont (View menü) 57, 81
Project menü Delphi 32 42, 54, 59
Project menüpont (Options menü) 83
Project Options dialógusablak 206
Project Source menüpont (View menü) 57, 81
Projekt 23
  kezelése 252
public kulcsszó 276, 281
PWideChar típus 153

R

raise kulcsszó (kivételek előidézése) 301, 302
Range Checking fordítási direktíva 258
ReadOnly tulajdonság (TEdit komponens) 118
Real típus 150
Rebuild Library menüpont (Component menü) Configure Palette menüpont (Component menü) 60
Redo menüpont (Edit menü) 54, 78
Refresh metódus 187

Rekordok 224
Rekurzió 249
Relációs adatbázis 411
Remove File from Project ikon (eszközpaletta) 45, 69
Remove File menüpont (File menü) 77
Remove From Project menüpont (File menü) 52
Rename (fájlkezelés) 322
ReOpen menüpont (File menü) 52
Repaint metódus 187
repeat ciklus 179
Replace menüpont (Search menü) 56, 80
RequestData metódus (DDEClientConv) 357
Reset művelet (fájlkezelés) 333
Részletablak 290, 292
Rewrite művelet (fájlkezelés) 333
RmDir (fájlkezelés) 322
RowCount tulajdonság (TStringGrid, TDrawGrid komponens) 127
RTF formátum 403
RTL 238
Run
  hibakeresés 308
  ikon (eszközpaletta) 44, 69
  menü Delphi 32 60
Run to Cursor (hibakeresés) 308
Runtime Library 238
  rutinok 238

S, Sz

Save All
  ikon (eszközpaletta) 44
  menüpont (File menü) 52
Save As menüpont (File menü) 52
Save File As menüpont (File menü) 76
Save File from Project ikon (eszközpaletta) 45
Save File
  ikon (eszközpaletta) 45, 69
  menüpont (File menü) 76
Save
  ikon (Eszközpaletta) 69
  menüpont (File menü) 52
Save Project As menüpont (File menü) 52, 76
Save Project menüpont (File menü) 75
Scale menüpont (Edit menü) 55, 80
ScaleBy metódus 187
Scaled tulajdonság 100
Search Again menüpont (Search menü) 56, 81
Search menü
  Delphi 16 80
  Delphi 32 42, 56
SelAttributes tulajdonság (TRichEdit komponens) 407
Select All menüpont (Edit menü) 55, 79
Select Form from List ikon (eszközpaletta) 44, 69
Select Unit from List ikon (eszközpaletta) 44, 69
self kulcsszó 288
Send to Back
  menüpont (Edit menü) 55, 79
  metódus 186
Sender paraméter 282
ServerConv tulajdonság (DDEServerItem) 353
ServiceApplication tulajdonság (DDEServerConv) 355
SetLink metódus (DDEClientConv) 357
ShortCut tulajdonság (menük) 132
ShortInt típus 149
ShortString típus 154
Show Execution Point (hibakeresés) 308
Show Hints fordítási direktíva 260
Show metódus 186, 207
Show Warnings fordítási direktíva 260
ShowHints tulajdonság 97
ShowMessage eljárás 194
ShowMessagePos eljárás 194
ShowModal metódus 208
Single típus 150
Size menüpont (Edit menü) 55, 79
Smart Callbacks fordítási direktíva 261
Sorszámozott típus 212
  műveletek 213
Sorted tulajdonság (TListBox, TComboBox komponens) 123
Spacing tulajdonság (TBitButton komponens) 122

Speedbar menüpont (View menü) 59, 83
Stack Checking fordítási direktíva 258
Standard oldal (komponenspaletta) 105
Step Over
  hibakeresés 308, 313
  ikon (eszközpaletta) 45, 69
StrDispose eljárás 234
Stretch tulajdonság (TImage komponens) 375
Strict Var-Strings fordítási direktíva 258
String típus 151
Strings tulajdonság (sztringlisták) 221
StrNew eljárás 234
StrToFloat függvény (típuskonverziók) 174
StrToInt függvény (típuskonverziók) 174
Style tulajdonság
  grafika 378, 378
  TListBox, TComboBox komponens 123
Súgórendszer 49, 74, 403
Syntax Check menüpont (Project menü) 59
System oldal (komponenspaletta) 105
Számok 169
  műveletei 169, 170
Színek (grfika) 376
Szövegszerkesztés 52, 78
Sztringek 167
Sztringlisták 220
Szülőform (MDI) 390
Szülőkomponens 110

T

Tab Order menüpont (Edit menü) 55, 80
Tag tulajdonság 101
TApplication objektum 187
Tartalmazó komponens 110
TBatchMove komponens (Data Access komponensek) 414
TBevel komponens 107
TBitButton komponens 107, 120
TButton komponens 106, 120
TCanvas objektum 190
TCheckBox komponens 106, 124
TColorDialog dialógusablak 386, 388
TColorDialog komponens 108
TComboBox komponens 106, 122
TDataBase komponens (Data Access komponensek) 414
TDataSource komponens (Data Access komponensek) 414, 416
TDBCheckBox komponens (Data Controls komponensek) 417
TDBComboBox komponens (Data Controls komponensek) 417
TDBCtrlGrid komponens (Data Controls komponensek Delphi 32) 418
TDBEdit komponens (Data Controls komponensek) 417
TDBGrid komponens (Data Controls komponensek) 417
TDBImage komponens (Data Controls komponensek) 417
TDBListBox komponens (Data Controls komponensek) 417
TDBLookUpCombo komponens (Data Controls komponensek Delphi 16) 418
TDBLookupComboBox komponens (Data Controls komponensek Delphi 32) 418
TDBLookupList komponens (Data Controls komponensek Delphi 16) 418
TDBLookupListBox komponens (Data Controls komponensek Delphi 32) 418
TDBMemo komponens (Data Controls komponensek) 417
TDBNavigator komponens (Data Controls komponensek) 417
TDBRadioGroup komponens (Data Controls komponensek) 417
TDBText komponens (Data Controls komponensek) 417
TDDEClient komponens (DDE) 350
TDDEClientConv komponens 109
  DDE 350, 356
TDDEClientItem komponens 109
TDDEServerConv komponens 109
  DDE 350, 355
TDDEServerItem komponens 109
  DDE 350, 353, 354
TDirectoryListBox komponens 108
TDrawGrid komponens 107, 127
TDriveComboBox komponens 108

TDW 311
TEdit komponens 106, 118
Terminate metódus (TApplication objektum) 187
Text tulajdonság
  DDEClientItem 358
  DDEServerItem 354
  TEdit komponens 119
TextFile kulcsszó (fájlkezelés) 332
TextHeight metódus (TCanvas objektum) 191
TextOut metódus (TCanvas objektum) 190, 191
TextWidth metódus (TCanvas objektum) 191
TFileListBox komponens 108
TFilterComboBox komponens 108
TFindDialog
  dialógusablak 386
  komponens 108
TFontDialog
  dialógusablak 386, 389
  komponens 108
TGroupBox komponens 106, 110
THeader komponens 107
THeaderControl komponens (Windows 95 vezérlőelemek) 407
THotKey komponens (Windows 95 vezérlőelemek) 407
Threads menüpont (View menü) 58
TImage komponens 107
  grafika 371, 375
TImageList komponens (Windows 95 vezérlőelemek) 407
Típus 148, 211
  sorszámozott 212
Típus nélküli állományok (fájlkezelés) 341
Típuskompatibilitás 158
Típuskonverziók 173
Típusos állományok (fájlkezelés) 339
Title tulajdonság (TOpenDialog, TSaveDialog komponens) 388
TLabel komponens 106, 117
TListBox komponens 106, 122
TListView komponens (Windows 95 vezérlőelemek) 407
TMainMenu komponens 106
TMaskEdit komponens 107
TMediaPlayer komponens 109
TMemo komponens 106
TMemo komponens 119
TNoteBook komponens 107, 110
Többalakúság 270
TObject osztály 273, 291
Toggle Form/Unit ikon (eszköz-paletta) 45, 69
Toggle Form/Unit menüpont (View menü) 58, 83
TOLEContainer komponens 109, 359, 362, 364
  tulajdonságai 365
Tömbök 216
  átadása 247
  egydimenziós 217
  többdimenziós 223
Tools
  menü (Delphi 16) 85
  menü (Delphi 32) 42, 61
  menüpont (Tools menü) 62
Top tulajdonság 99
ToPage tulajdonság (TPrint komponens) 390
TOpenDialog
  dialógusablak 386
  komponens 108
TOutline komponens 107
TPageControl komponens (Windows 95 vezérlőelemek) 407
TPaintBox komponens 108
TPanel komponens 106, 110, 382
TPopUpMenu komponens 106
TPrint dialógusablak 390
TPrintDialog
  dialógusablak 386
  komponens 108
TPrinter objektum 191
TPrinterSetupDialog
  dialógusablak 386
  komponens 108
TProgressBar komponens (Windows 95 vezérlőelemek) 407

TQuery komponens (Data Access komponensek) 414, 415
Trace Into
  hibakeresés 308, 313
  ikon (eszközpaletta) 45, 69
TRadioButton komponens 106, 124
TRadioGroup komponens 106, 110
Transparent tulajdonság (TLabel komponens) 117
TReplaceDialog
  dialógusablak 386
  komponens 108
TReport komponens (Data Access komponensek) 414
TRichEdit komponens (Windows 95 vezérlőelemek) 407, 408
try kulcsszó 233, 233
try/except blokk 294, 301
try/finally blokk 287
TSaveDialog
  dialógusablak 386
  komponens 108
TScreen objektum 189
TScrollBar komponens 106
TScrollBox komponens 107,110
TSession komponens (Data Access komponensek) 414
TShape komponens 107
  grafika 371, 372
TShiftState paraméterek (egéresemények) 397
TSpeedButton komponens 107
TStatusBar komponens (Windows 95 vezérlőelemek) 407
TStoredProc komponens (Data Access komponensek) 414
TStringGrid komponens 107, 127
TStrings objektum 220
TTabbedNote komponens 107, 110
TTabControl komponens (Windows 95 vezérlőelemek) 407
TTable komponens (Data Access komponensek) 414, 415
TTabSet komponens 107
TThread komponens (Delphi 32) 344
TTimer komponens 108, 124
TTrackBar komponens (Windows 95 vezérlőelemek) 407
TTreeView komponens (Windows 95 vezérlőelemek) 407
Tulajdonság 27
  beállítása 144
TUpDateSQL komponens (Data Access komponensek) 414
TUpDown komponens (Windows 95 vezérlőelemek) 407
Turbo Debugger for Windows 311
Typed @ operátor fordítási direktíva 259

U

Undelete menüpont (Edit menü) 54, 78
Undo menüpont (Edit menü) 54, 78
Unitok
  kezelése 252
  részei 253
Units menüpont (View menü) 58,83
Use Unit menüpont (File menü) 52
uses kulcsszó 202, 203, 245

Ü

Üzenetablak 194
fajtái 196

V

Vágólap 346
Valós típusok 150
Változó 156
  dinamikus 165
  statikus 165
  unit-szintű 165
Változótípusok 148
var
  blokk 164
  kulcsszó 161
Variant típus 155, 212
VCL 271
Védett
  blokkok 294
  erőforrásfoglalások 302

VertScroll tulajdonság 98
View menü
  Delphi 16 81
  Delphi 32 42, 57
Visible tulajdonság 101, 115
Visual Component Library 271, 280

W

WantReturns tulajdonság (TMemo komponens) 120
WantsTabs tulajdonság (TMemo komponens) 120
Watch funkció (hibakeresés) 315
Watches
  hibakeresés 307
  menüpont (View menü) 58, 82
WideChar típus 153
Width tulajdonság 97, 115
  grafika 378
  TScreen objektum 190
  TStringGrid, TDrawGrid komponens 127
Window List menüpont (View menü) 58, 82
Windows 95 vezérlőelemek
Windows API 240
Windows help-kezelő 405
Windows Stack Frame fordítási direktíva 261
WindowState tulajdonság 101
with kulcsszó 147
Word Align Data fordítási direktíva 260
Word típus 150
WordWrap tulajdonság (TLabel komponens) 117

Z

Zárójelezés 172
Zártság 268
Z-sorrend 381, 392

Egyetemi Nyomda — 97.4629 Budapest, 1997
Felelős vezető: Sümeghi Zoltán igazgató

Kezdő Haladó ✔ Felsőoktatási tankönyv ✔

A Delphi a Microsoft Windows-alapú alkalmazások fejlesztésének rendkívül sokoldalú eszköze. Magán a programnyelven kívül a Borland cég látványos (első látásra talán egy repülőgép műszerfalához hasonló) és könnyen kezelhető környezettel igyekszik a programozók kegyét kivívni magának – úgy tűnik, sikerrel. Erősségei közé tartozik az a modern fordítótechnika, amellyel hihetetlenül gyors, önmagában (értelmező rendszer nélkül) futtatható programokat hozhatunk létre. Lehetőséget kínál valódi objektumorientált programozásra is. Népszerűségének egyik fontos tényezője, hogy rendelkezik a kliens-szerver adatbázis fejlesztéséhez szükséges eszközökkel is.

A Delphi megtartotta alapként a Pascal kellemes tulajdonságait, miközben több területen is újításokat tartalmaz: objektumorientáltság, kivételkezelés (a hibák strukturált feldolgozását biztosító eszköz), eseményvezéreltség stb. Egyik nagy előnye, hogy a felhasználói felület interaktív módon tervezhető a fejlesztői környezetből: például nyomógombokat rakosgathatunk tetszés szerinti helyre, menüket helyezhetünk el. A programozónak ilyen módon tényleg csak arra kell koncentrálnia, hogy a vizuális elemekhez tartozó funkciókat implementálja.

Lépésről lépésre szinte észrevétlenül juttatja a tanulni vágyókat a megfelelő tudásanyag birtokába.

A Programozók Könyvtára sorozat következő kötetei:

Visual Basic – Lépésről lépésre
Visual Basic 4 – Tippek és trükkök

ISBN 9635451245
9 789635 451241

Ára: 1990,– Ft